

MATLAB Programming for Finite Element Methods

Terence YUE Yu

MATLAB version: R2018a

目 录

1 一维问题的有限元方法	1
1.1 单元与整体的关系	1
1.1.1 整体节点基与局部节点基	1
1.1.2 整体刚度矩阵与单元刚度矩阵的关系	2
1.2 刚度矩阵与载荷向量的装配	4
1.2.1 载荷向量的装配 — 局部与整体对应	4
1.2.2 刚度矩阵的装配	5
1.2.3 装配的编程实现	6
1.3 边界项的处理与程序整理	8
1.3.1 边界项的处理	8
1.3.2 程序整理	12
1.4 sparse 装配法	15
1.4.1 刚度矩阵的装配	16
1.4.2 载荷向量的装配	19
1.4.3 程序整理	22
1.5 基于变分形式的编程	24
1.5.1 变分形式的计算	25
1.5.2 双线性形式的程序设计	27
1.5.3 程序整理	30
2 网格的图示与标记	33
2.1 网格与解的图示	33
2.1.1 基本数据结构	33
2.1.2 补片函数 patch	34
2.1.3 操作 cell 数组的 cellfun 函数	35
2.1.4 showmesh 函数的建立	36
2.1.5 showsolution 函数的建立	37
2.2 网格的标记	41
2.2.1 节点标记	41
2.2.2 单元标记	41
2.2.3 边的标记	43

3 辅助数据结构与几何量	49
3.1 辅助数据结构	49
3.1.1 elem2edge 的生成	49
3.1.2 edge2elem 的生成	53
3.1.3 neighbor 的生成	55
3.2 网格相关的几何量	56
3.3 auxstructure 与 auxgeometry 函数	57
3.4 边界设置	59
3.4.1 边界边的定向	60
3.4.2 边界的设置	61
4 二维问题的有限元方法	64
4.1 问题描述与网格数据	64
4.2 二维问题的装配	67
4.2.1 指标装配法	67
4.2.2 sparse 装配法	72
4.3 边界积分的处理	75
4.3.1 等效拉平法	75
4.3.2 边界积分的装配	77
4.3.3 边界单元的积分计算	78
4.3.4 边界条件的程序实现	79
4.3.5 解的图示	80
4.4 二维问题程序整理	81
5 自适应有限元方法	87
5.1 自适应方法简介	87
5.1.1 后验误差估计	87
5.1.2 加密或标记准则	88
5.1.3 有限元程序	89
5.2 误差指示子的计算	91
5.2.1 残差的计算	91
5.2.2 边界跳量的计算	92
5.2.3 indicator 函数	95
5.3 标记算法的实现	97

5.4	Newest-node bisection	99
5.4.1	局部加密方式	99
5.4.2	最新点二分的简单说明	101
5.4.3	标记二分的程序实现	103
5.4.4	协调二分的程序实现	106
5.4.5	Newest-node bisection 程序整理	107
5.5	自适应有限元程序	109
6	线弹性边值问题	112
6.1	线弹性边值问题简介	112
6.1.1	问题说明	112
6.1.2	连续变分问题	113
6.1.3	有限元方法	114
6.2	刚度矩阵与载荷向量的装配	115
6.2.1	单元的向量法分析	115
6.2.2	sparse 装配指标	116
6.2.3	双线性分量的配对	118
6.3	第一种形式的变分问题	120
6.3.1	刚度矩阵的计算	120
6.3.2	载荷向量的计算	122
6.3.3	Neumann 边界条件	124
6.3.4	Dirichlet 边界条件	125
6.3.5	程序整理	126
6.4	第二种形式的变分问题	130
6.5	第三种形式的变分问题	131
7	Kirchhoff 板弯问题	137
7.1	平衡方程、变分问题与边界条件	137
7.1.1	平衡方程与边界条件	137
7.1.2	变分问题	138
7.2	非协调 Morley 元	138
7.2.1	Morley 元的构造	138
局部节点基	138	
整体节点基	140	

7.2.2	sparse 装配指标	142
7.2.3	刚度矩阵与载荷向量的计算	143
	双线性形式第一项的计算	143
	双线性形式第二项的计算	147
	载荷向量的计算	148
7.2.4	边界条件的处理	148
7.2.5	数值结果	149
7.3	非协调 Zienkiewicz 元	150
7.3.1	Zienkiewicz 元的构造	150
7.3.2	sparse 装配指标	152
7.3.3	双线性形式第一项的计算	152
8	多重网格法的算法介绍	156
8.1	嵌套有限元	156
8.1.1	嵌套有限元空间与子空间方程	156
8.1.2	延长、限制算子与延长、限制矩阵	157
8.1.3	Galerkin 条件	158
8.2	MG 的基本思想	159
8.2.1	残差校正与频率分量	159
8.2.2	经典迭代法对频率分量的影响	160
8.2.3	两网格方法	162
8.3	MG 的算法描述	164
8.3.1	MG 的两网格添加	164
8.3.2	V-循环的伪代码	166
8.4	MG 矩阵的获得	167
8.4.1	延长矩阵	167
8.4.2	延长矩阵的程序实现	169
8.5	一维问题的 MG 方法	171
8.5.1	刚度矩阵和载荷向量	171
8.5.2	多重网格函数	173
	mgVcycle1D 函数	173
	Vcycle1D 函数	174
	smoother 函数	175

第一章 一维问题的有限元方法

以如下的混合边值问题作为本章的模型问题

$$\begin{cases} -u'' + cu = f(x), & 0 < x < 1, \\ u(0) = 0, \quad u'(1) = 0, \end{cases} \quad (1.1)$$

其中 $c > 0$ 是常数. 变分问题为: 找 $u \in V$, 使得

$$a(u, v) = \ell(v) \quad \forall v \in V, \quad (1.2)$$

式中,

$$\begin{aligned} a(u, v) &= \int_0^1 (u'v' + cuv)dx, \quad \ell(v) = (f, v) = \int_0^1 fvdx, \\ V &= \{v \in L^2(0, 1) : a(v, v) < \infty, \quad v(0) = 0\}. \end{aligned}$$

1.1 单元与整体的关系

有限元编程中最重要的就是如何从单元刚度矩阵和单元载荷向量获得整体刚度矩阵和整体载荷向量, 我们称这个过程为装配. 为了实现装配, 首先必须弄清楚单元刚度矩阵或载荷向量与整体刚度矩阵或载荷向量有什么关系.

有限元方法是基于变分形式的数值方法, 在乘以检验函数并分部积分时, 就不可避免地要遇到边界项. 例如, 对模型问题 (1.1), 在不考虑边界条件的情况下, 变分形式应该为

$$\int_0^1 (u'v' + cuv)dx - u'|_0^1 = \int_0^1 f(x)v(x)dx.$$

对高维问题, $-u'|_0^1$ 通常对应边界积分. 要注意, 对不同的边界条件, 边界积分可能含有未知量, 对模型问题 (1.1), 它恰好为零. 为了方便, 我们统称边界积分项和边界条件为边界项, 而且为了一般性, 有限元编程的一个重要原则是: 最后处理边界项. 为了突出省略了边界项, 变分形式写为

$$\int_0^1 (u'v' + cuv)dx \sim \int_0^1 f(x)v(x)dx,$$

符号 “ \sim ” 表示省略了边界项.

1.1.1 整体节点基与局部节点基

设所考虑问题的区域为 $\Omega = (0, 1)$, 给定划分 $0 = x_0 < x_1 < \dots < x_n = 1$, 设 $K_j = [x_{j-1}, x_j]$, $j = 1, 2, \dots, n$, 则可引入分段线性函数空间 (不包含边界条件)

$$V_h = \{v \in C(\bar{\Omega}) : v \text{ 在每个 } K_j = [x_{j-1}, x_j] \text{ 上连续且为一次函数}, j = 1, 2, \dots, n\}.$$

我们知道, 每个节点 x_i 都对应一个节点基 $\Phi_i(x)$, 满足 $\Phi_i(x_j) = \delta_{ij}$, $0 \leq i, j \leq n$, 它们合起来构成空间 V_h 的一组基. 这些基的图像如下图所示

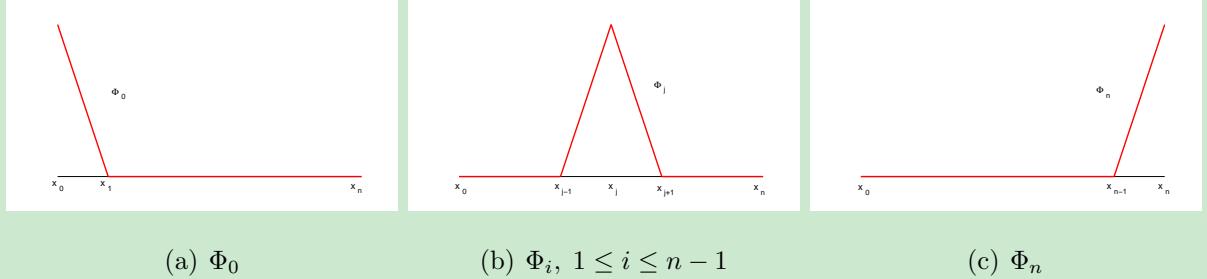


图 1.1. 整体基函数 $\Phi_i, i = 0, 1, \dots, n$

如果把整个区域 $\bar{\Omega}$ 换成单元 $K = [x_{j-1}, x_j]$, 那么可得两个节点基 ϕ_1, ϕ_2 , 其图像如下图所示

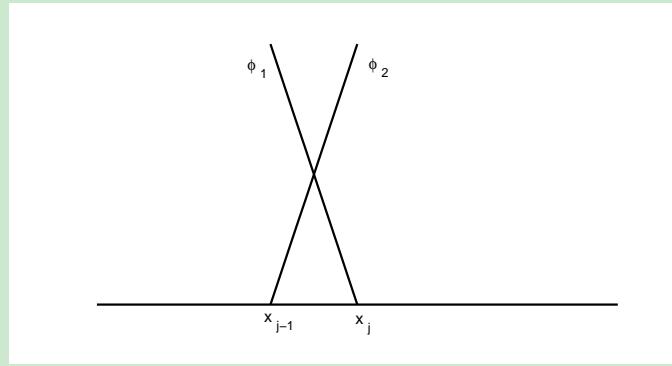


图 1.2. 单元 K 的节点基

我们称 ϕ_1, ϕ_2 为局部节点基.

一个重要的关系就是, 整体节点基限制在单元上就是局部节点基. 例如,

$$\Phi_{j-1}|_K = \phi_1, \quad \Phi_j|_K = \phi_2.$$

之所以成立, 是因为 $\Phi_{j-1}|_K$ 是一次函数, 且 $\Phi_{j-1}|_K(x_{j-1}) = 1$, $\Phi_{j-1}|_K(x_j) = 0$. 这个关系对高维问题仍成立, 正是有这一条性质才导致有限元可以很方便地按单元考虑.

1.1.2 整体刚度矩阵与单元刚度矩阵的关系

在不考虑边界项时, 整体刚度矩阵可由单元刚度矩阵获得, 我们以模型问题 (1.1) 来说明装配的过程.

步 1: 第 j 个方程及其单元分解

设整体基函数为 $\Phi_0, \Phi_1, \dots, \Phi_n$ (视 u_0 也为变量), 待求函数表为

$$u = \sum_{i=0}^n u_i \Phi_i,$$

则系统方程组的第 j ($= 0, 1, \dots, n$) 个方程就是在 (1.2) 中取 $v = \Phi_j$. 按单元求和, 左端和右端分别为

$$\begin{aligned} a(u, v) &= \int_0^1 (u'v' + cuv) dx = \sum_{i=1}^n \int_{x_{i-1}}^{x_i} (u'v' + cuv) dx, \quad v = \Phi_j, \\ \ell(v) &= \sum_{i=1}^n \int_{x_{i-1}}^{x_i} fv dx, \quad v = \Phi_j. \end{aligned}$$

为此, 我们可先考虑单元上的部分, 即

$$\begin{aligned} a(u, v)_{K_i} &= \int_{x_{i-1}}^{x_i} (u'v' + cuv) dx, \quad v = \Phi_j, \\ \ell(v)_{K_i} &= \int_{x_{i-1}}^{x_i} fv dx, \quad v = \Phi_j, \end{aligned}$$

它们是第 j 个方程的组成成分, 加起来的过程就是合并同类项.

步 2: 单元上的形式

设单元 K_i 上两个局部节点基为 ϕ_1 和 ϕ_2 , 则 u 限制在 K_i 就是 $u_{i-1}\phi_1 + u_i\phi_2$, 即

$$u = u_{i-1}\phi_1 + u_i\phi_2, \quad x \in K_i,$$

于是

$$a(u, v)_{K_i} = \int_{x_{i-1}}^{x_i} [(u_{i-1}\phi_1 + u_i\phi_2)'v' + c(u_{i-1}\phi_1 + u_i\phi_2)v] dx, \quad v = \Phi_j.$$

在逐个代入 $v = \Phi_j$ 时, 单元 K_i 上有贡献的只有 Φ_{i-1} 和 Φ_i , 局部上恰好对应 ϕ_1 和 ϕ_2 , 也就是说单元 K_i 上的积分实际上只在第 $i-1$ 个方程和第 i 个方程上有贡献, 它们分别为

$$\begin{aligned} v = \Phi_{i-1} : \quad \int_{x_{i-1}}^{x_i} [(u_{i-1}\phi_1 + u_i\phi_2)'\phi'_1 + c(u_{i-1}\phi_1 + u_i\phi_2)\phi_1] dx &\sim \int_{x_{i-1}}^{x_i} f\phi_1 dx, \\ v = \Phi_i : \quad \int_{x_{i-1}}^{x_i} [(u_{i-1}\phi_1 + u_i\phi_2)'\phi'_2 + c(u_{i-1}\phi_1 + u_i\phi_2)\phi_2] dx &\sim \int_{x_{i-1}}^{x_i} f\phi_2 dx, \end{aligned}$$

写成矩阵形式为

$$\int_{x_{i-1}}^{x_i} \begin{bmatrix} \phi'_1\phi'_1 + c\phi_1\phi_1 & \phi'_2\phi'_1 + c\phi_2\phi_1 \\ \phi'_1\phi'_2 + c\phi_1\phi_2 & \phi'_2\phi'_2 + c\phi_2\phi_2 \end{bmatrix} dx \begin{bmatrix} u_{i-1} \\ u_i \end{bmatrix} \sim \int_{x_{i-1}}^{x_i} \begin{bmatrix} f\phi_1 \\ f\phi_2 \end{bmatrix} dx.$$

可以看到左边的矩阵恰是单元刚度矩阵, 右边的向量恰是单元载荷向量. 注意矩阵的第一行对应 $v = \Phi_{i-1}$, 它应加到第 $i-1$ 个系统方程中, 第二行对应 $v = \Phi_i$, 它应加到第 i 个系统方程中, 与这里的变量 $[u_{i-1}, u_i]^T$ 正好对应.

步 3: 遍历所有单元

当遍历所有单元后, 就得到系统方程组, 而遍历的过程恰好是把单元刚度矩阵的分量加到整体刚度矩阵的对应位置.

1.2 刚度矩阵与载荷向量的装配

1.2.1 载荷向量的装配 — 局部与整体对应

我们已经看到, 装配的过程就是把单元刚度矩阵或载荷向量的元素加到整体刚度矩阵或载荷向量的正确位置, 这就需要局部与整体编号的一种对应.

为了方便, 我们先考虑载荷向量的装配. 设单元 $K_i = [x_{i-1}, x_i]$ ($i = 1, 2, \dots, n$) 上的两个局部节点基为 ϕ_1, ϕ_2 , 单元载荷向量为

$$F_{K_i} = \begin{bmatrix} \ell_{K_i}(\phi_1) \\ \ell_{K_i}(\phi_2) \end{bmatrix}.$$

对单元载荷向量, 可如下分析

a) ϕ_1 在 K_i 上对应的恰是左端点的整体节点基 Φ_{i-1} , ϕ_2 在 K_i 上对应的恰是右端点的整体节点基 Φ_i , 于是单元载荷向量用整体基函数表示为

$$F_{K_i} = \begin{bmatrix} \ell_{K_i}(\phi_1) \\ \ell_{K_i}(\phi_2) \end{bmatrix} = \begin{bmatrix} \ell_{K_i}(\Phi_{i-1}) \\ \ell_{K_i}(\Phi_i) \end{bmatrix}.$$

b) $\ell_{K_i}(\Phi_{i-1})$ 是 $\ell(\Phi_{i-1})$ 的一部分, 自然贡献给 $F_{i-1} = \ell(\Phi_{i-1})$;

$\ell_{K_i}(\Phi_i)$ 是 $\ell(\Phi_i)$ 的一部分, 自然贡献给 $F_i = \ell(\Phi_i)$.

c) 因此有如下贡献关系

$$F_{K_i} = \begin{bmatrix} \ell_{K_i}(\phi_1) \\ \ell_{K_i}(\phi_2) \end{bmatrix} \rightarrow \begin{bmatrix} F_{i-1} \\ F_i \end{bmatrix}.$$

综上, 我们可以获得如下的装配算法.

载荷向量的装配

1. 初始化载荷向量 F 为零向量, 注意行对应 u_0, u_1, \dots, u_n , 设单元编号 $i = 1$.
2. 计算相应的单元载荷 F_{K_i} .
3. 把单元向量的分量加入到整体载荷向量的对应位置

$$\begin{aligned} F_{i-1} &\leftarrow F_{i-1} + F_{K_i}(1), \\ F_i &\leftarrow F_i + F_{K_i}(2). \end{aligned}$$

4. $i \leftarrow i + 1$, 转到第 2 步, 直到 $i = n + 1$ 结束.
-

上面的装配过程表明有如下的局部与整体对应

index : $\{1, 2\}$ (local) \rightarrow $\{i-1, i\}$ (global), 对单元 K_i ,

即

$$\text{index}(1) = i - 1, \quad \text{index}(2) = i,$$

我们称 index 为装配指标. 显然, 局部整体对应与单元的连通性是一致的.

1.2.2 刚度矩阵的装配

当 x_i, x_j 不相邻时, $K_{ij} = 0$, 故 K 是一个稀疏矩阵(且对称). 设单元 $K_i = [x_{i-1}, x_i]$ ($i = 1, 2, \dots, n$) 的两个节点基为 ϕ_1, ϕ_2 , 对应的单元刚度矩阵为

$$A_{K_i} = \begin{bmatrix} a(\phi_1, \phi_1)_{K_i} & a(\phi_1, \phi_2)_{K_i} \\ a(\phi_2, \phi_1)_{K_i} & a(\phi_2, \phi_2)_{K_i} \end{bmatrix}.$$

对单元刚度矩阵, 如下分析

a) ϕ_1 在 K_i 上对应的恰是 Φ_{i-1} , ϕ_2 在 K_i 上对应的恰是 Φ_i , 于是单元刚度矩阵用整体基函数表示为

$$A_{K_i} = \begin{bmatrix} a(\phi_1, \phi_1)_{K_i} & a(\phi_1, \phi_2)_{K_i} \\ a(\phi_2, \phi_1)_{K_i} & a(\phi_2, \phi_2)_{K_i} \end{bmatrix} = \begin{bmatrix} a(\Phi_{i-1}, \Phi_{i-1})_{K_i} & a(\Phi_{i-1}, \Phi_i)_{K_i} \\ a(\Phi_i, \Phi_{i-1})_{K_i} & a(\Phi_i, \Phi_i)_{K_i} \end{bmatrix}.$$

b) $a(\Phi_{i-1}, \Phi_{i-1})_{K_i}$ 是 $a(\Phi_{i-1}, \Phi_{i-1})$ 的一部分, 自然贡献给 $K_{i-1,i-1} = a(\Phi_{i-1}, \Phi_{i-1})$;

$a(\Phi_{i-1}, \Phi_i)_{K_i}$ 是 $a(\Phi_{i-1}, \Phi_i)$ 的一部分, 自然贡献给 $K_{i-1,i} = a(\Phi_{i-1}, \Phi_i)$;

$a(\Phi_i, \Phi_{i-1})_{K_i}$ 是 $a(\Phi_i, \Phi_{i-1})$ 的一部分, 自然贡献给 $K_{i,i-1} = a(\Phi_i, \Phi_{i-1})$;

$a(\Phi_i, \Phi_i)_{K_i}$ 是 $a(\Phi_i, \Phi_i)$ 的一部分, 自然贡献给 $K_{i,i} = a(\Phi_i, \Phi_i)$.

c) 因此有如下贡献关系

$$A_{K_i} = \begin{bmatrix} a(\phi_1, \phi_1)_{K_i} & a(\phi_1, \phi_2)_{K_i} \\ a(\phi_2, \phi_1)_{K_i} & a(\phi_2, \phi_2)_{K_i} \end{bmatrix} \rightarrow \begin{bmatrix} K_{i-1,i-1} & K_{i-1,i} \\ K_{i,i-1} & K_{i,i} \end{bmatrix}.$$

我们有如下装配算法.

刚度矩阵的装配

1. 初始化刚度矩阵 K 为零矩阵, 注意行对应 u_0, u_1, \dots, u_n , 设单元编号 $i = 1$.
2. 计算相应的单元刚度矩阵 A_{K_i} .
3. 把单元矩阵的分量加入到整体刚度矩阵的对应位置

$$\begin{aligned} K_{i-1,i-1} &\leftarrow K_{i-1,i-1} + A_{K_i}(1,1), \\ K_{i-1,i} &\leftarrow K_{i-1,i} + A_{K_i}(1,2), \\ K_{i,i-1} &\leftarrow K_{i,i-1} + A_{K_i}(2,1), \\ K_{i,i} &\leftarrow K_{i,i} + A_{K_i}(2,2). \end{aligned}$$

4. $i \leftarrow i + 1$, 转到第 2 步, 直到 $i = n + 1$ 结束.
-

注意到

$$\begin{bmatrix} K_{i-1,i-1} & K_{i-1,i} \\ K_{i,i-1} & K_{i,i} \end{bmatrix}$$

恰好在矩阵三对角位置上, 因此整体刚度矩阵是三对角的. 一般而言, 单元刚度矩阵不是对称的, 若是对称的, 则只要计算上三角部分.

1.2.3 装配的编程实现

鉴于 MATLAB 向量标号是从 1 开始, 以下也遵循这个规定, 并给出如下的 5 个线性单元剖分.

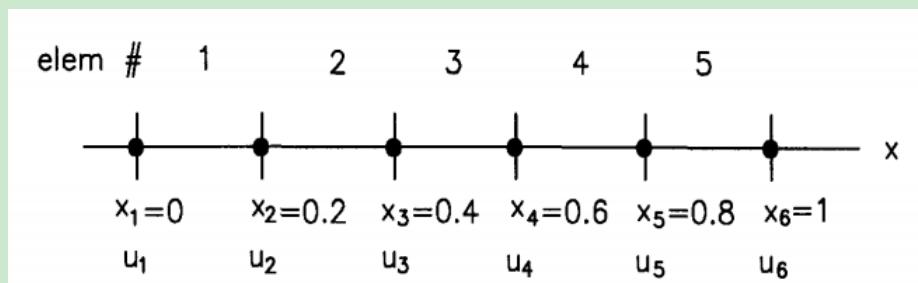


图 1.3. 5 个线性单元的剖分

设单元方程如下

$$\begin{bmatrix} k_{11}^i & k_{12}^i \\ k_{21}^i & k_{22}^i \end{bmatrix} \begin{bmatrix} u_i \\ u_{i+1} \end{bmatrix} \sim \begin{bmatrix} F_i \\ F_{i+1} \end{bmatrix},$$

局部上, u_i 和 u_{i+1} 用 1 和 2 对应, 整体上则用 i 和 $i + 1$ 表示, 即

$$\{1, 2\} \text{ (local)} \quad \rightarrow \quad \{i, i + 1\} \text{ (global)},$$

于是可建立如下数组

$$\text{index}(1) = i, \quad \text{index}(2) = i + 1, \quad \text{对第 } i \text{ 个单元},$$

它实现了局部与整体的对应.

装配分两步进行, 即装配单元矩阵和单元向量. 对每个单元, 用 **ke** 表示单元矩阵, **fe** 表示单元向量. 用 **kk** 表示系统矩阵, **ff** 表示系统向量. 设单元矩阵和单元向量如下

$$\mathbf{ke} = \begin{bmatrix} \frac{1}{h_i} + \frac{h_i}{3} & -\frac{1}{h_i} + \frac{h_i}{6} \\ -\frac{1}{h_i} + \frac{h_i}{6} & \frac{1}{h_i} + \frac{h_i}{3} \end{bmatrix}, \quad \mathbf{fe} = \begin{bmatrix} \frac{h_i}{6}(2x_i + x_{i+1}) \\ \frac{h_i}{6}(2x_{i+1} + x_i) \end{bmatrix} \longleftrightarrow \begin{bmatrix} u_i \\ u_{i+1} \end{bmatrix}.$$

我们先装配单元向量. **fe** 的第一行对应系统的 i , 即 $\text{index}(1) = i$, **fe** 的第二行对应系统的 $i + 1$, 即 $\text{index}(2) = i + 1$. **f** 的第一行应加到 **ff** 的第 i 行, 第二行应加到 **ff** 的第 $i + 1$ 行. 为此可如下编写程序:

```

1 for iel = 1:nel          % 单元循环
2     index = elem(iel,:);% local-->global
3     for i = 1:2           % f 的行循环 (local)
4         ii = index(i);   % f 的行在系统中的编号 (global)
5         ff(ii) = ff(ii)+fe(i); % 把 f 的行放到系统向量的行中
6     end
7 end

```

ke 既涉及到行又涉及到列, 在前面装配 **fe** 的时候实际上已经确定了行, 现在需要确定列. 注意与 u_j 相乘的元素位于第 j 列, 即与指标 **index** 对应.

为此, 装备 **ke** 和 **fe** 的过程可如下编写程序:

```

1 for iel = 1:nel          % 单元循环
2     index = elem(iel,:); % local-->global
3     for i = 1:2           % f 的行循环 (local)
4         ii = index(i);   % f 的行在系统中的编号 (global)
5         ff(ii) = ff(ii)+fe(i); % 把 f 的行放到系统向量的行中
6         for j = 1:2         % k 的列循环 (local)
7             jj = index(j); % k 的列在系统中的编号 (global)

```

```

8         kk(ii,jj) = kk(ii,jj)+ke(i,j);
9         % 把 k 的行列元素放到系统向量的行列中
10        end
11    end
12 end

```

MATLAB 支持矩阵用向量取多行或多列, 因而相加的过程可简单写为

```

1 ff(index) = ff(index)+fe;
2 kk(index,index) = kk(index,index)+ke;

```

这里 $kk(index, index)$ 包含交叉位置的元素. 本文称这种装配策略为指标法, 它的好处在于明确了局部与整体的对应, 便于直接推广. 事实上, 上面的装配程序可以直接平移到高阶有限元以及高维问题中, 只不过要相应修改 $index$ 罢了, 后面将会看到这一点.

注 1.1 上面的装配过程适合逐个单元进行, 即一边计算单元刚度矩阵和载荷向量, 一边进行装配. 一种更高效的装配策略是, 我们首先计算出所有单元刚度矩阵, 然后用 `sparse` 函数进行一次性装配, 后面说明.

在求解方程组时, 我们可把矩阵转为稀疏矩阵, 即 $kk = \text{sparse}(kk)$. 对单元数很多时, 这样处理会节省较多时间.

1.3 边界项的处理与程序整理

1.3.1 边界项的处理

这一节说明为什么边界项可以最后处理. 我们以一个具体的例子重述前面的过程.

考虑方程

$$-u'' + u = x, \quad 0 < x < 1,$$

这里先不管边界条件. 对应的变分形式为

$$a(u, v) = \ell(v),$$

其中

$$a(u, v) = \int_0^1 (u'v' + uv) dx, \quad \ell(v) = \int_0^1 xv dx + u'|_0^1.$$

设区间划分为 $0 = x_0 < x_1 < \cdots < x_n = 1$, 记单元 $K_j = [x_{j-1}, x_j]$, $j = 1, 2, \dots, n$, $h_j = x_j - x_{j-1}$. 其上的插值基函数为 (统一记下标为 1,2)

$$\phi_1(x) = \frac{x_j - x}{h_j}, \quad \phi_2(x) = \frac{x - x_{j-1}}{h_j}.$$

变分形式写成单元累加形式为

$$\sum_{i=1}^n \int_{x_{i-1}}^{x_i} (u'v' + uv)dx = \sum_{i=1}^n \int_{x_{i-1}}^{x_i} xvdx + u'v|_0^1. \quad (1.3)$$

先考虑

$$I = \sum_{i=1}^n \int_{x_{i-1}}^{x_i} (u'v' + uv)dx - \sum_{i=1}^n \int_{x_{i-1}}^{x_i} xvdx,$$

单元 K_i 对应的是

$$I_i = \int_{x_{i-1}}^{x_i} (u'v' + uv)dx - \int_{x_{i-1}}^{x_i} xvdx.$$

把插值近似 $u = u_{i-1}\phi_1 + u_i\phi_2$ 和 $v = \phi_1, \phi_2$ 分别代入上式, 经过简单计算有离散形式

$$\tilde{I}_i = \begin{bmatrix} \frac{1}{h_{i-1}} + \frac{h_{i-1}}{3} & -\frac{1}{h_{i-1}} + \frac{h_{i-1}}{6} \\ -\frac{1}{h_{i-1}} + \frac{h_{i-1}}{6} & \frac{1}{h_{i-1}} + \frac{h_{i-1}}{3} \end{bmatrix} \begin{bmatrix} u_{i-1} \\ u_i \end{bmatrix} - \begin{bmatrix} \frac{h_{i-1}}{6}(2x_{i-1} + x_i) \\ \frac{h_{i-1}}{6}(2x_i + x_{i-1}) \end{bmatrix}.$$

若现在只有三个等分单元, 即 $x_0 = 0, x_1 = 1/3, x_2 = 2/3$ 和 $x_3 = 1$, 则对单元 1, 有

$$\tilde{I}_1 = \begin{bmatrix} 3.111 & -2.9444 \\ -2.9444 & 3.111 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \end{bmatrix} - \begin{bmatrix} 0.0185 \\ 0.0370 \end{bmatrix},$$

对单元 2, 有

$$\tilde{I}_2 = \begin{bmatrix} 3.111 & -2.9444 \\ -2.9444 & 3.111 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} - \begin{bmatrix} 0.0741 \\ 0.0926 \end{bmatrix},$$

对单元 3, 有

$$\tilde{I}_3 = \begin{bmatrix} 3.111 & -2.9444 \\ -2.9444 & 3.111 \end{bmatrix} \begin{bmatrix} u_2 \\ u_3 \end{bmatrix} - \begin{bmatrix} 0.1296 \\ 0.1481 \end{bmatrix}.$$

局部到整体的装配, 写出来就是如下的自然扩展: 对单元 1, 有

$$\tilde{I}_1 = \begin{bmatrix} 3.111 & -2.9444 & 0 & 0 \\ -2.9444 & 3.111 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} - \begin{bmatrix} 0.0185 \\ 0.0370 \\ 0 \\ 0 \end{bmatrix},$$

对单元 2, 有

$$\tilde{I}_2 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 3.111 & -2.9444 & 0 \\ 0 & -2.9444 & 3.111 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} - \begin{bmatrix} 0 \\ 0.0741 \\ 0.0926 \\ 0 \end{bmatrix},$$

对单元 3, 有

$$\tilde{I}_3 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 3.111 & -2.9444 \\ 0 & 0 & -2.9444 & 3.111 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0.1296 \\ 0.1481 \end{bmatrix}.$$

三个扩展相加即得需要的整体刚度矩阵和载荷向量

$$\begin{bmatrix} 3.111 & -2.9444 & 0 & 0 \\ -2.9444 & 6.2222 & -2.9444 & 0 \\ 0 & -2.9444 & 6.2222 & -2.9444 \\ 0 & 0 & -2.9444 & 3.1111 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} 0.0185 \\ 0.1111 \\ 0.2222 \\ 0.1481 \end{bmatrix}.$$

式 (1.3) 的右端还有积分产生的边界项 $u'v|_0^1$, 现在考察它的贡献. 第 j 个方程是取 $v = \Phi_j$, 因此该方程的右端要加上

$$\begin{aligned} v = \Phi_j : \quad u'v|_0^1 &= u'\Phi_j|_0^1 = u'(1)\Phi_j(x_n) - u'(0)\Phi_j(x_0) \\ &= u'(1)\delta_{jn} - u'(0)\delta_{j0}. \end{aligned}$$

显然只需要在第 0 行加上 $-u'(0)$, 最后一行加上 $u'(1)$, 最终得到

$$\begin{bmatrix} 3.111 & -2.9444 & 0 & 0 \\ -2.9444 & 6.2222 & -2.9444 & 0 \\ 0 & -2.9444 & 6.2222 & -2.9444 \\ 0 & 0 & -2.9444 & 3.1111 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} 0.0185 - u'(0) \\ 0.1111 \\ 0.2222 \\ 0.1481 + u'(1) \end{bmatrix}. \quad (1.4)$$

最后考虑边界条件的处理. 若给出的是 Dirichlet 边界条件 $u(0) = u(1) = 0$, 则理论上我们选择的检验函数空间要求 $v(0) = v(1) = 0$, 从而 (1.3) 中的 $u'v|_0^1$ 自动消除. 对非齐次情形可通过边界条件齐次化转化为齐次情形, 但此时方程会发生变化, 变分形式相应地有所改变. 我们真正处理时并不是这样做的, 而是把第一行和最后一行分别用恒等式 $u_0 = u(0)$ 和 $u_n = u(1)$ 代替, 对这里的例子就是

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ -2.9444 & 6.2222 & -2.9444 & 0 \\ 0 & -2.9444 & 6.2222 & -2.9444 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} u(0) \\ 0.1111 \\ 0.2222 \\ u(1) \end{bmatrix}.$$

这种做法有一个缺点, 就是破坏了矩阵的对称性. 为此可以把已知节点值移到右端

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 6.2222 & -2.9444 & 0 \\ 0 & -2.9444 & 6.2222 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} u(0) \\ 0.1111 + 2.9444u_0 \\ 0.2222 + 2.9444u_3 \\ u(1) \end{bmatrix},$$

其中 $u_0 = u(0)$, $u_3 = u(1)$. 为了降低矩阵的规模, 可考虑去除恒等的行, 即

$$\begin{bmatrix} 6.2222 & -2.9444 \\ -2.9444 & 6.2222 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} 0.1111 + 2.9444u_0 \\ 0.2222 + 2.9444u_3 \end{bmatrix}.$$

以后都采用这种处理. 类似可处理其他边界条件, 例如, 当左边界是 Dirichlet 边界条件, 而右边界是 Neumann 边界条件时, 则只要把 (1.4) 的第一行恒等替换, 而最后一行代入 Neumann 边界值即可.

注 1.2 以后规定: 先处理 Neumann 边界条件, 最后处理 Dirichlet 边界条件. 这一点对高维问题比较重要. 例如对矩形区域, 设上边界为 Neumann, 右边界为 Dirichlet. 若先处理 Dirichlet, 再处理 Neumann, 则右上角的点可能变成未知点 (除非人为记住, 这样没有前者方便).

一维问题的边界条件非常容易处理, 只在区间的端点处, 程序如下 (总假设有 Dirichlet 边界条件)

```

1 Neumann = 1; Dirichlet = N;
2 g_N = du(node(Neumann));
3 g_D = uexact(node(Dirichlet));
4
5 % ----- Neumann boundary conditions -----
6 if ~isempty(Neumann)
7     bn = [-1; zeros(N-2,1); 1]; % -1: left norm vector
8     bn(Neumann) = bn(Neumann)*g_N;
9     ff = ff + (-1)*bn; % -u'
10 end
11
12 % ----- Dirichlet boundary conditions -----
13 isBdNode = false(N,1); isBdNode(Dirichlet) = true;
14 bdNode = find(isBdNode); freeNode = find(~isBdNode);
15 u = zeros(N,1); u(bdNode) = g_D;
16 ff = ff - kk*u;

```

这里,

- Line 1: Neumann 和 Dirichlet 边界条件的整体编号.
- Line 2-3: 边界值.
- Line 7: 分部积分涉及到左右边界的法向量.
- Line 14: `bdNode` 是 Dirichlet 边界编号 (可直接用 `bdNode = Dirichlet`), 而 `freeNode` 是未知变量的编号.

1.3.2 程序整理

例 1.1 考虑更一般的两点边值问题

$$au'' + bu' + cu = f(x), \quad 0 < x < L,$$

取精确解为

$$u(x) = \frac{1}{2e} e^{2x} - \frac{1}{2}(1 + e^{-1})e^x + \frac{1}{2}.$$

边界条件可以是 Dirichlet 边界条件或 Neumann 边界条件.

变分形式为

$$a(u, v) = \ell(v),$$

其中

$$\begin{aligned} a(u, v) &= \int_0^L (-au'v' + bu'v + cuv)dx, \\ \ell(v) &= \int_0^L f(x)v(x)dx - au'v|_0^L. \end{aligned} \tag{1.5}$$

单元刚度矩阵为

$$\begin{aligned} [K^e] &= \int_{x_i}^{x_{i+1}} \left(-a \begin{bmatrix} \phi'_1 \\ \phi'_2 \end{bmatrix} [\phi'_1, \phi'_2] + b \begin{bmatrix} \phi_1 \\ \phi_2 \end{bmatrix} [\phi'_1, \phi'_2] + c \begin{bmatrix} \phi_1 \\ \phi_2 \end{bmatrix} [\phi_1, \phi_2] \right) dx \\ &= -\frac{a}{h_i} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} + \frac{b}{2} \begin{bmatrix} -1 & 1 \\ -1 & 1 \end{bmatrix} + \frac{ch_i}{6} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}, \end{aligned} \tag{1.6}$$

单元向量为

$$[F^e] = \int_{x_i}^{x_{i+1}} f(x) \begin{bmatrix} \phi_1 \\ \phi_2 \end{bmatrix} dx \approx \frac{f(x_c)h_i}{2} \begin{bmatrix} 1 \\ 1 \end{bmatrix},$$

其中 x_c 是单元 $K_i = [x_i, x_{i+1}]$ 的中点, 即 $x_c = (x_i + x_{i+1})/2$.

在后面编程中, 我们主要用到如下的数据信息.

1. 节点编号及坐标

我们将用 `node` 表示所有节点的坐标, `node` 的索引即为节点编号.

2. 连通性

连通性指的是单元的顶点编号, 我们用 `elem` 表示, 它的第一列表示所有单元的第一个顶点编号, 第二列是第二个顶点编号. 由连通性立刻可以获得局部与整体对应的装配指标 `index`.

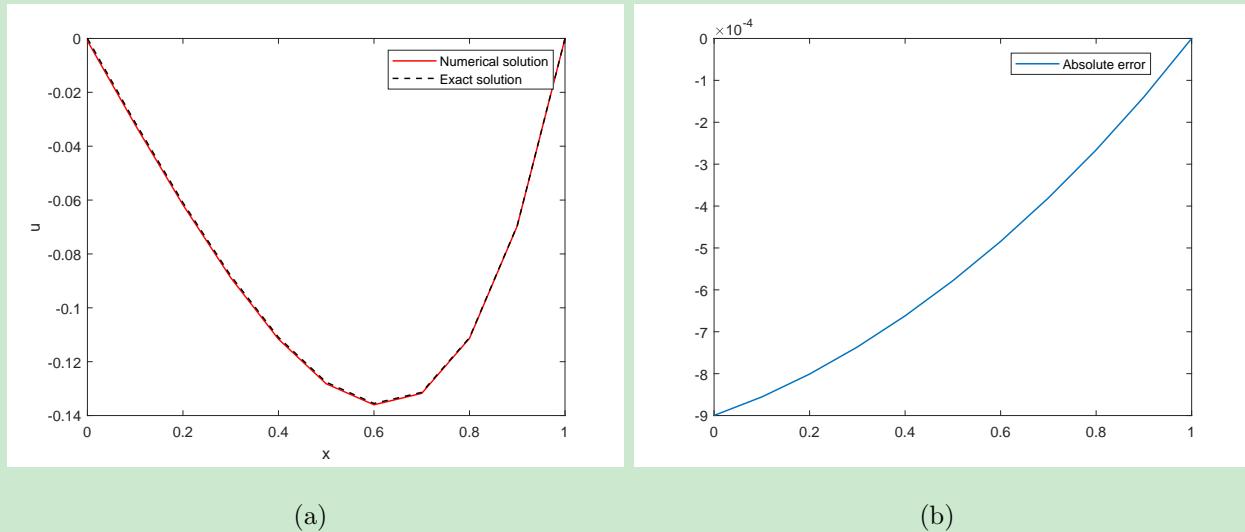


图 1.4. (a) 数值解和精确解; (b) 绝对误差 ($n = 10$)

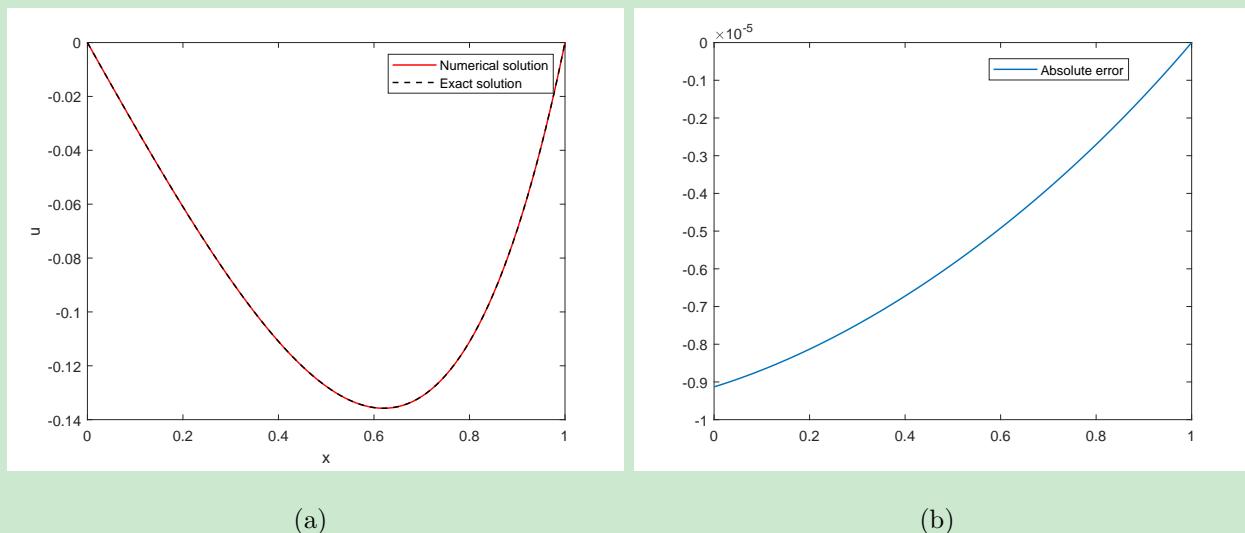


图 1.5. (a) 数值解和精确解; (b) 绝对误差 ($n = 100$)

程序如下

CODE 1.1. FEM1D.m

```
1 clc;clear; close all;
2 tic;
3 % ----- Mesh -----
4 a = 0; b = 1;
5 nel = 10; N = nel+1; % numbers of elements and nodes
6 node = linspace(a,b,nel+1)';
7 elem = zeros(nel,2); elem(:,1) = 1:N-1; elem(:,2) = 2:N;
8
9 % ----- PDE -----
10 acoef = 1; bcoef = 1; ccoef = 1;
11 syms x;
12 c1 = 0.5/exp(1); c2 = -0.5*(1+1/exp(1));
13 u = c1*exp(2*x)+c2*exp(x)+1/2;
14 uexact = eval(['@(x)',vectorize(u)]); % transform to anonymous ...
    function
15
16 du = diff(u);
17 du = eval(['@(x)',vectorize(du)]);
18
19 f = acoef*diff(u,2)+bcoef*diff(u,1)+ccoeff*u;
20 f = eval(['@(x)',vectorize(f)]);
21
22 Neumann = 1; Dirichlet = N;
23 g_N = du(node(Neumann));
24 g_D = uexact(node(Dirichlet));
25
26 % ----- Assembly -----
27 kk = zeros(N,N); ff = zeros(N,1);
28 for iel = 1:nel
    % local --> global
29     index = elem(iel,:);
30     nl = index(1); nr = index(2);
31     xl = node(nl); xr = node(nr); he = xr-xl;
32
33     % element matrix and vector
34     a1 = -(acoef/he); a2 = bcoef/2; a3 = ccoef*he/6;
35     ke = a1*[1 -1;-1 1]+a2*[-1 1;-1 1]+a3*[2 1;1 2];
36     fe = f((xl+xr)/2)*[he/2;he/2];
```

```

38
39 % assemble
40 kk(index,index) = kk(index,index)+ke;
41 ff(index) = ff(index)+fe;
42 end
43 kk = sparse(kk);
44
45 % ----- Neumann boundary conditions -----
46 if ~isempty(Neumann)
47 bn = [-1;zeros(N-2,1);1]; % -1: left norm vector
48 bn(Neumann) = bn(Neumann)*g_N;
49 ff = ff + (-acoef)*bn;
50 end
51
52 % ----- Dirichlet boundary conditions -----
53 isBdNode = false(N,1); isBdNode(Dirichlet) = true;
54 bdNode = find(isBdNode); freeNode = find(~isBdNode);
55 u = zeros(N,1); u(bdNode) = g_D;
56 ff = ff - kk*u;
57
58 % ----- error analysis -----
59 uexact = uexact(node);
60 u(freeNode) = kk(freeNode,freeNode)\ff(freeNode);
61 figure, plot(node,u,'r-o',node,uexact,'k-','linewidth',1)
62 xlabel('x'); ylabel('u')
63 legend('Numerical solution','Exact solution')
64 Err = u-uexact;
65 figure, plot(node,Err,'linewidth',1)
66 legend('Absolute error')
67 toc

```

1.4 sparse 装配法

前面给出的装配为

```

1 kk(index,index) = kk(index,index)+ke;
```

它易于理解和推广, 但未充分利用 MATLAB 的向量运算. 事实上, 我们可以把所有的指标对应放在一起实现一次性装配, 本文称为 `sparse` 装配法. 注意, 它只是前面指标装配的再加工.

1.4.1 刚度矩阵的装配

考虑 (1.6) 给出的单元刚度矩阵. 为了实现一次性装配, 我们要用 k_{ij} 存储所有单元 (i, j) 位置的结果, 这里的 (i, j) 是局部编号, 即

```
1 % ----- All element matrices -----
2 h = diff(node);
3 k11 = -acoef./h+bcoef/2*(-1)+ccoef*h./6*2;
4 k12 = -acoef./h*(-1)+bcoef/2+ccoef*h./6;
5 k21 = -acoef./h*(-1)+bcoef/2*(-1)+ccoef*h./6;
6 k22 = -acoef./h+bcoef/2+ccoef*h./6*2;
```

根据对称性, 上面只需存储上三角部分, 为了一般性, 这里全部存储.

设单元的左端点整体编号为 z_1 , 右端点为 z_2 , 我们要给出所有单元的 local-global 对应, 显然为

```
1 % ----- local --> global -----
2 z1 = elem(:,1); z2 = elem(:,2);
```

且前面给出的第 i_{el} 个单元的装配指标就是

```
index = [z1(iel), z2(iel)].
```

我们可按照下面的方式逐一装配.

```
1 % ----- Assemble the matrix -----
2 % upper triangular
3 kk = sparse(z1,z2,k12,N,N);
4 % lower triangular
5 kk = kk+sparse(z2,z1,k21,N,N);
6 % diagonal
7 kk = kk+sparse(z1,z1,k11,N,N);
8 kk = kk+sparse(z2,z2,k22,N,N);
```

如果是对称矩阵, 那么可如下编写

```
1 kk = sparse(z1,z2,k12,N,N);
2 kk = kk+kk';
3 kk = kk+sparse(z1,z1,k11,N,N);
4 kk = kk+sparse(z2,z2,k22,N,N);
```

这个装配方法比之前的指标法更快一点, 但仍有缺点:

1. 对高维问题, 单元刚度矩阵的分量较多, 逐一相加比较麻烦;

2. `sparse` 函数的快速在于建立稀疏矩阵, 做加减等运算时, 它还是要按稠密矩阵进行计算.

在优化之前, 我们先来分析一下上面的装配.

- 对固定单元, 设单元刚度矩阵为

$$A_K = \begin{array}{cc|c} u_i & u_{i+1} \\ \hline k_{11} & k_{12} & u_i \\ k_{21} & k_{22} & u_{i+1} \end{array},$$

这里右侧表示整体刚度矩阵的行指标, 上侧表示列指标. 对 k_{12} , 我们要把它放在 $(i, i + 1)$ 位置, 按照前面编程的思路, 是 (z_1, z_2) 位置. 也就是说, 我们要在 (z_1, z_2) 处赋值 k_{12} , 这可用稀疏矩阵函数 `sparse` 完成, 其用法如下

```
S = sparse(i, j, s, m, n);
```

它分配了一个 $m \times n$ 的稀疏零矩阵, 其中 i, j 是向量, 对应分量指出非零值的位置, s 则是非零值的向量. 因此, 若想在 (z_1, z_2) 处赋值 k_{12} , 可以如下操作

```
A = sparse(z1, z2, k12, N, N);
```

注意, 该命令把所有单元刚度矩阵的 $(1,2)$ 处的值放到了整体刚度矩阵的对应位置中.

- 我们自然会有这样的疑问: 如果有两个单元刚度矩阵 $(1,2)$ 处的值对应相同的整体指标 (i, j) , 那么它们应该相加, 而不是简单的赋值.

事实上, 在 MATLAB 中, `sparse` 函数有一个特殊性质 (summation property): 当出现相同指标 (i, j) 时, 规定非零值相加. 这样, 若的确出现上面的情况, 则本身已经解决.

我们要说的是, 对非对角线元素, 上面提到的情形是不可能出现的. 因为 K_i 对应 $\{u_i, u_{i+1}\}$, 只可能出现 $(i, i + 1), (i + 1, i)$, 而对 K_{i+1} 则为 $(i + 1, i + 2), (i + 2, i + 1)$, 两者没有共同位置.

- 上面最后处理对角线元素是防止对称情形转置相加时重复相加对角线元素.

`sparse` 函数的 summation property 使得我们可以进一步优化上面的装配过程, 即把所有位置的 (i, j, s) 拼接在一起实现, 即

$$\begin{bmatrix} i_{11} & j_{11} & s_{11} \\ i_{12} & j_{12} & s_{12} \\ i_{21} & j_{21} & s_{21} \\ i_{22} & j_{22} & s_{22} \end{bmatrix},$$

称其为 sparse 装配指标.

1. 我们按单元刚度矩阵行优先的顺序排列每列的指标;
2. \mathbf{i}_{ij} 表示所有单元矩阵 (i, j) 位置的整体编号向量, 共有 nel 个元素, 其中 nel 是单元的个数;
3. 记上面的矩阵为 $[\mathbf{i}, \mathbf{j}, \mathbf{s}]$, 则 \mathbf{i} 共有 $\text{nel} \times N_{dof}^2$ 个元素, 其中, $N_{dof} = 2$ 表示单元的自由度个数.

核心代码如下

```

1 % ----- local --> global -----
2 Ndof = 2; nnz = nel*Ndof^2;
3 ii = zeros(nnz,1); jj = zeros(nnz,1); ss = zeros(nnz,1);
4 id = 0;
5 for i = 1:Ndof
6     for j = 1:Ndof
7         ii(id+1:id+nel) = elem(:,i);    % zi
8         jj(id+1:id+nel) = elem(:,j);    % zj
9         ss(id+1:id+nel) = K(:,i,j);    % kij
10        id = id + nel;
11    end
12 end

```

这里假设 (i, j) 位置的单元刚度矩阵值用三维数组存储, 如 iFEM.

注 1.3 关于上面的装配我们做如下评注:

1. `elem` 按行存储单元的好处是可以按列操作同一个顶点.
2. 可用向量法给出 `ii`, `jj`, 考虑到只是简单的赋值, 这里不这样处理, 以体现直观.
3. 我们将采用另一种方式存储单元矩阵, 而不是像 iFEM 那样用三维数组. 做法非常简单, 即按行拉直存储所有单元的结果

$$K = [k_{11}, k_{12}, k_{21}, k_{22}],$$

这里, k_{11} 是所有单元给出的向量, 类似其他. 显然 `ss` 由该矩阵按列拉直得到. **后面将采用这种处理, 为此在生成单元刚度矩阵时, 我们实施行拉直.**

我们可以如下给出装配算法

```

1 % local --> global
2 K = [k11,k12,k21,k22]; % stored in rows
3 Ndof = 2; nnz = nel*Ndof^2;
4 ii = zeros(nnz,1); jj = zeros(nnz,1); ss = zeros(nnz,1);
5 id = 0; s = 1;
6 for i = 1:Ndof
7     for j = 1:Ndof
8         ii(id+1:id+nel) = elem(:,i); % zi
9         jj(id+1:id+nel) = elem(:,j); % zj
10        ss(id+1:id+nel) = K(:,s); % kij
11        id = id + nel; s = s+1;
12    end
13 end
14 % stiffness matrix
15 kk = sparse(ii,jj,ss,N,N);

```

或

CODE 1.2. sparse 装配

```

1 % local --> global
2 K = [k11,k12,k21,k22];
3 Ndof = 2; nnz = nel*Ndof^2;
4 ii = zeros(nnz,1); jj = zeros(nnz,1); ss = K(:, );
5 id = 0;
6 for i = 1:Ndof
7     for j = 1:Ndof
8         ii(id+1:id+nel) = elem(:,i); % zi
9         jj(id+1:id+nel) = elem(:,j); % zj
10        id = id + nel;
11    end
12 end
13 % stiffness matrix
14 kk = sparse(ii,jj,ss,N,N);

```

1.4.2 载荷向量的装配

单元载荷向量为

$$[F_K] = \begin{bmatrix} F_1 \\ F_2 \end{bmatrix}, \quad F_i = \int_K f v dx dy, \quad v = \phi_i, \quad i = 1, 2,$$

用中心格式近似有

$$[F^e] = \int_K \begin{bmatrix} f\phi_1 \\ f\phi_2 \end{bmatrix} dx dy \approx \begin{bmatrix} f\phi_1 \\ f\phi_2 \end{bmatrix}_{x_c} \cdot |K| = f(x_c) \cdot \frac{h_i}{2} \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

显然有

$$F_1 = F_2 = f(x_c) \cdot \frac{h_i}{2}.$$

按前面逐一装配的方式可如下进行

```
1 % ----- Assemble the vector -----
2 x1 = node(1:N-1); x2 = node(2:N); xc = (x1+x2)./2;
3 fi = f(xc).*h./2;
4 F1 = fi; F2 = fi;
5 ff = sparse(z1,1,F1,N,1);
6 ff = ff+sparse(z2,1,F2,N,1);
```

根据 `sparse` 的 summation property, 最后的两句实际上可直接写为

```
1 ff = sparse([z1;z2],1,[F1;F2],N,1);
```

注意到 $[z1; z2] = \text{elem}(:)$, 也可写为

```
1 ff = sparse(elem(:,1),[F1;F2],N,1);
```

`ff` 一般不是稀疏的, 存储为稀疏的形式访问起来会麻烦. 我们可用下面的语句代替

```
1 ff = accumarray(elem(:,1),[F1;F2],[N 1]);
```

同理, 右端向量也按行拉直存储, 即

```
1 % ----- Assemble the vector -----
2 x1 = node(1:N-1); x2 = node(2:N);
3 xc = (x1+x2)./2;
4 F1 = f(xc).*h./2; F2 = F1; F = [F1,F2];
5 ff = accumarray(elem(:,1), F(:,1), [N 1]);
```

`accumarray` 是累加函数, 用法如下.

1. 若 `accumarray` 有两个参数, 则第一个参数是数组的位置索引, 第二个参数是累加的数据. 例如

```
1 subs = [1; 2; 4; 2; 4]; vals = 101:105;
2 A = accumarray(subs,vals);
```

`subs` 必须是列向量, 其中的最大数为 4 表明 `A` 是 4 个元素的数组且初始为零数组; 索引 1 对应的值为 101, 则 `A` 的第 1 个位置累加 101; 索引 2 对应的有两个, 分别为 102, 104, 它们累加起来为 206; 索引 3 没有则仍为 0; 索引 4 对应有 103, 105, 累加起来为 208. 于是 `A = [101; 206; 0; 208].`

特别地, 当 `vals = 2` 为单个数时, 默认为 `vals = vals*ones(max(subs), 1)`, 此时 `A = [2; 4; 0; 4].`

2. 位置索引可以是矩阵. 例如

```

1 subs = [1 1;
2      2 1;
3      2 3;
4      2 1;
5      2 3];
6 vals = 101:105
7 A = accumarray(subs,vals);

```

`subs` 列的最大值分别为 2,3, 则 `A` 是 2×3 的矩阵; `subs` 的行与 `vals` 的行对应; `subs` 的第一行是 (1,1) 表示矩阵的该位置累加 101, 第二行是 (2,1) 表示矩阵的该位置累加 102. 根据这个规律, 我们有 `A = [101 0 0; 206 0 208].`

再考虑三维数组的例子.

```

1 subs = [1 1 1;
2      2 1 2;
3      2 3 2;
4      2 1 2;
5      2 3 2];
6 vals = 101:105;
7 A = accumarray(subs,vals);

```

`subs` 列的最大值分别为 2,3,3, 则 `A` 是 $2 \times 3 \times 2$ 的矩阵; 第一行为 (1,1,1), 则矩阵的 (1,1,1) 累加 101, 类似其他行. 这样, 我们有

```

A(:,:,1) =
101      0      0
      0      0      0

A(:,:,2) =
      0      0      0

```

3. 上面是通过列的最大值来确定矩阵 A 的维数, 我们也可自行设定矩阵的维数. 例如

```

1 subs = [1 1;
2      2 1;
3      2 3;
4      2 1;
5      2 3];
6 vals = 1;
7 A = accumarray(subs, vals, [2 4]);

```

若没有第三个参数 [2 4] (必须是行向量), 则 A 是 2×3 的矩阵 $[1 \ 0 \ 0; 2 \ 0 \ 2]$. 加上 [2 4], 则是 2×4 的矩阵, 分析一致, 此时结果为 $[1 \ 0 \ 0 \ 0; 2 \ 0 \ 2 \ 0]$.

根据上面的分析, `ff = accumarray(elem(:,), F(:,), [N 1])` 是生成一个 $N \times 1$ 的零向量 ff, 且在 `elem(i)` 的位置累加 `F(i)`.

1.4.3 程序整理

考虑例 1.1 的问题, 数值结果与前面相同, 程序如下.

CODE 1.3. FEM1D_Sparse.m

```

1 clc;clear; close all;
2 tic;
3 % ----- Mesh -----
4 a = 0; b = 1;
5 nel = 10; N = nel+1; % numbers of elements and nodes
6 node = linspace(a,b,nel+1)';
7 elem = zeros(nel,2); elem(:,1) = 1:N-1; elem(:,2) = 2:N;
8
9 % ----- PDE -----
10 acoef = 1; bcoef = 1; ccoef = 1;
11 syms x;
12 c1 = 0.5/exp(1); c2 = -0.5*(1+1/exp(1));
13 u = c1*exp(2*x)+c2*exp(x)+1/2;
14 % exact solution
15 uexact = eval(['@(x)',vectorize(u)]); % transform to anonymous ...
function

```

```

16
17 du = diff(u);
18 du = eval(['@(x)',vectorize(du)]);
19
20 % rhs
21 f = acoef*diff(u,2)+bcoef*diff(u,1)+ccoeff*u;
22 f = eval(['@(x)',vectorize(f)]);
23
24 % boundary conditions
25 Neumann = 1; Dirichlet = N;
26 g_N = du(node(Neumann));
27 g_D = uexact(node(Dirichlet));
28
29 % ----- Assemble the matrix -----
30 % All element matrices
31 h = diff(node);
32 k11 = -acoef./h+bcoef/2*(-1)+ccoeff*h./6*2;
33 k12 = -acoef./h*(-1)+bcoef/2+ccoeff*h./6;
34 k21 = -acoef./h*(-1)+bcoef/2*(-1)+ccoeff*h./6;
35 k22 = -acoef./h+bcoef/2+ccoeff*h./6*2;
36 K = [k11,k12,k21,k22]; % stored in rows
37
38 % local --> global
39 Ndof = 2; nnz = nel*Ndof^2;
40 ii = zeros(nnz,1); jj = zeros(nnz,1); ss = K(:);
41 id = 0; % s = 1;
42 for i = 1:Ndof
43     for j = 1:Ndof
44         ii(id+1:id+nel) = elem(:,i); % zi
45         jj(id+1:id+nel) = elem(:,j); % zj
46         % ss(id+1:id+nel) = K(:,s); % kij
47         id = id + nel; % s = s+1;
48     end
49 end
50
51 % stiffness matrix
52 kk = sparse(ii,jj,ss,N,N);
53
54 % ----- Assemble the vector -----
55 x1 = node(1:N-1); x2 = node(2:N);

```

```

56 xc = (x1+x2)./2;
57 F1 = f(xc).*h./2; F2 = F1; F = [F1,F2];
58 ff = accumarray(elem(:), F(:,[N 1]));
59
60 % ----- Neumann boundary conditions -----
61 if ~isempty(Neumann)
62     bn = [-1;zeros(N-2,1);1]; % -1: left norm vector
63     bn(Neumann) = bn(Neumann)*g_N;
64     ff = ff + (-acoef)*bn;
65 end
66
67 % ----- Dirichlet boundary conditions -----
68 isBdNode = false(N,1); isBdNode(Dirichlet) = true;
69 bdNode = find(isBdNode); freeNode = find(~isBdNode);
70 u = zeros(N,1); u(bdNode) = g_D;
71 ff = ff - kk*u;
72
73 % ----- error analysis -----
74 uexact = uexact(node);
75 u(freeNode) = kk(freeNode,freeNode)\ff(freeNode);
76 figure, plot(node,u,'r-',node,uexact,'k--','linewidth',1)
77 xlabel('x'); ylabel('u')
78 legend('Numerical solution','Exact solution')
79 Err = u-uexact;
80 figure, plot(node,Err,'linewidth',1)
81 legend('Absolute error')
82 toc

```

1.5 基于变分形式的编程

FreeFem++ 是一款非常优秀的有限元分析软件, 它的一大亮点是编程过程与变分形式一一对应, 本文称为“基于变分形式的编程”. 这种处理方式通常使用面向对象的语言, 程序组织以及阅读起来都非常困难. 笔者曾一度想学习面向对象编程, 但最终还是放弃了. 一方面对 C++ 不是很熟悉, 另一方面也没有太多精力. 对有限元编程过程的再思考之后, 我觉得也可以用面向过程的语言写出基于变分形式的程序. 其实本质在于处理各种典型的双线性形式以及线性形式.

1.5.1 变分形式的计算

对一维问题, 双线性形式一般是如下典型项的组合

$$\int_{\Omega} au'v' dx, \quad \int_{\Omega} auvw dx, \quad \int_{\Omega} au'vdx, \quad \int_{\Omega} auv'dx,$$

这里 Ω 是区间, a 可以是函数. 以最后一项为例. 通常习惯把检验函数 v 的项放在前面, 即令

$$a(u, v) = \int_{\Omega} auv' dx = \int_{\Omega} av'u dx,$$

这是因为单元刚度矩阵为 (线性元)

$$[K^e] = \int_K a \begin{bmatrix} \phi_1 \\ \phi_2 \end{bmatrix}' [\phi_1, \phi_2] dx = \int_K a \begin{bmatrix} \phi'_1 \phi_1 & \phi'_1 \phi_2 \\ \phi'_2 \phi_1 & \phi'_2 \phi_2 \end{bmatrix} dx,$$

顺序上一致.

设

$$v_1 = \phi'_1, \quad v_2 = \phi'_2, \quad u_1 = \phi_1, \quad u_2 = \phi_2,$$

则

$$[K^e] = \int_K a \begin{bmatrix} v_1 u_1 & v_1 u_2 \\ v_2 u_1 & v_2 u_2 \end{bmatrix} dx =: \begin{bmatrix} k_{11} & k_{12} \\ k_{21} & k_{22} \end{bmatrix},$$

式中,

$$k_{ij} = \int_K av_i u_j dx, \quad K = [x_l, x_r],$$

且积分可采用 Simpson 公式计算. 为了一般性我们采用 Gauss 积分公式, 且把积分转化到参考元上.

设参考元为 $[0, 1]$, 对区间 $[a, b]$, 定义仿射变换

$$x \in [a, b] \mapsto t \in [0, 1],$$

即

$$t = \frac{x - a}{b - a} \quad \text{或} \quad x = (b - a)t + a.$$

令 $\tilde{w}(t) = w(x(t))$, 则

$$\begin{aligned} w(x) &= \tilde{w}(t) = c_1 \tilde{w}(t) \quad (c_1 = 1), \\ w'(x) &= \tilde{w}'(t) \frac{dt}{dx} = \frac{1}{b - a} \tilde{w}'(t) =: c_2 \tilde{w}'(t), \\ dx &= (b - a)dt =: J dt, \end{aligned}$$

且

$$k_{ij} = \int_K a v_i u_j dx = \int_K a \phi'_i \phi_j dx = c_1 c_2 J \cdot \int_{[0,1]} \tilde{a} \tilde{\phi}'_i \tilde{\phi}_j dt.$$

在参考元 $[0, 1]$ 上, 两个基函数分别为

$$\tilde{\phi}_1(t) = 1 - t, \quad \tilde{\phi}_2(t) = t,$$

且变换前后 $a(x)$ 与 $\tilde{a}(t)$ 在对应点处的值相同. 设 $[0, 1]$ 上的 Gauss 点为 r_0, \dots, r_N , 权重为 w_0, \dots, w_N , 则任给区间 $[x_l, x_r]$ 上的 Gauss 点为

$$x_i = x(r_i) = x_l + r_i h, \quad h = x_r - x_l, \quad r \in I = [-1, 1],$$

相应的权重为

$$w_i = \frac{h}{2} w_i, \quad i = 0, 1, \dots, N.$$

这样, 积分可如下计算.

- 系数函数 $\tilde{a}(t)$ 在 $[0, 1]$ 上的 Gauss 点与 $a(x)$ 在每个单元上的 Gauss 点在仿射变化下对应. 这样, 所有单元的 Gauss 点如下获得

```

1 [r, w] = GaussQuad(4, 0, 1);
2 xa = node(elem(:, 1)); xb = node(elem(:, 2));
3 xx = xa+0.5*(xb-xa)*(1+r');
% quadrature points on all elements

```

这里, r, w 是参考元上的 Gauss 点和权重, 而 xx 是每个单元上的 Gauss 点 (每行对应一个单元).

- 系数函数对应的函数值为

```

1 cf = cf(xx);

```

它的每行对应一个单元, 且是变换到参考元上的函数值. 而基函数在参考元上的函数值为

```

1 v1 = v1(r); v2 = v2(r); u1 = u1(r); u2 = u2(r);

```

它们都只是一列的数据. 显然 cf 的每列与 $v1$ 的每行对应. 于是积分为

```

1 ke = w.*[v1.*u1, v1.*u2, v2.*u1, v2.*u2];
2 Kref = cf*ke;

```

注意, 这里刚度矩阵按行拉直排列的 (根据前面装配的说明).

- 上面给出的是变换到参考元上的结果, 还原时需要乘以变换系数

```
1 K = cv.*cu.*J.*Kref;
```

其他双线性形式的计算只需要适当修改 v_1, v_2, u_1, u_2 即可.

类似可给出右端线性形式的计算. 对一维问题, Neumann 边界条件比较简单, 不把它看作线性形式, 这样右端只可能为

$$\ell(v) = \int_{\Omega} f v dx.$$

```
1 cf = f(xx); fe = w.*[v1(r),v2(r)]; Fref = cf*fe;
2 cv = Trans1D(v,xa,xb); F = cv.*J.*Fref;
3 ff = accumarray(elem(:), F(:), [N 1]);
```

1.5.2 双线性形式的程序设计

前面已经提到, 程序中只需要修改 v_1, v_2, u_1, u_2 即可. 考虑变分形式 (1.5), 我们给出如下对应

```
1 Coef = {-acoef, bcoef, ccoef};
2 Trial = {'u.dx', 'u.dx', 'u.val'};
3 Test = {'v.dx', 'v.val', 'v.val'};
```

这里, u 对应试探函数 (即数值解对应的函数), v 对应检验函数; $u.val$ 表示函数本身, $u.dx$ 表示一阶导. 注意同一位置的三个数据对应双线性形式的一个组份. 双线性形式的函数格式如下

```
1 kk = Bilinear1D(Th,Coef,Trial,Test);
```

这里 Th 存储 node , elem .

通过检测试探函数或检验函数的类型给出参考基函数, 如下

```
1 function [w1,w2] = Refbase1D(w)
2 % Trial or Test functions on reference interval [0,1]
3 if strcmpi(w,'u.val') || strcmpi(w,'v.val')
4     w1 = @(x) 1-x; w2 = @(x) x;
5 elseif strcmpi(w,'u.dx') || strcmpi(w,'v.dx')
6     w1 = @(x) -ones(size(x)); w2 = @(x) ones(size(x));
7 else
8     error("The input character must be 'u.val', 'u.dx', 'v.val' ...
or 'v.dx'.");
```

```
9     end  
10 end
```

这里, `strcmpi(w, 'u.dx')` 是字符比较(不区分大小写), 当检测到 w 是 '`u.dx`' 或 '`v.dx`' 时, 返回的 `w1, w2` 为相应基函数的导数.

设 `xa, xb` 分别表示所有单元的左右节点坐标, 则不同类型的检验或试探函数在变换下产生的系数如下获取

```
1 function cw = Trans1D(w,xa,xb)  
2 % coefficients of the affine transformation  
3 if strcmpi(w,'u.val') || strcmpi(w,'v.val')  
    nel = length(xa); cw = ones(nel,1);  
5 elseif strcmpi(w,'u.dx') || strcmpi(w,'v.dx')  
    cw = 1./(xb-xa);  
7 else  
    error("The input character must be 'u.val', 'u.dx', 'v.val' ...  
          or 'v.dx'.");  
9 end  
10 end
```

变换的 Jacobian 为

```
1 function J = Jacobian1D(xa,xb)  
2 J = xb-xa;  
3 end
```

有了上面的准备, 所有单元的刚度矩阵可如下获得

```
1 K = zeros(nel,2*Ndof); % straighten for easy assembly  
2 J = Jacobian1D(xa,xb); % Jacobian on all elements  
3 [r, w] = GaussQuad(4,0,1); % on [0,1];  
4 xx = xa+(xb-xa)*r'; % quadrature points on all elements  
5 for s = 1:length(Coef) % s-th component of bilinear form  
6     u = Trial{s}; v = Test{s};  
7     % Trial and Test functions  
8     [u1,u2] = Refbase1D(u); [v1,v2] = Refbase1D(v);  
9     % Stiffness matrix on reference interval [0,1]  
10    cf = Coef{s};  
11    if isa(cf,'double'), cf = @(x) cf+0*x; end  
12    cf = cf(xx);  
13    v1 = v1(r); v2 = v2(r); u1 = u1(r); u2 = u2(r);
```

```

14     ke = w.*[v1.*u1, v1.*u2, v2.*u1, v2.*u2];
15     Kref = cf*ke;
16 % Stiffness matrix on all elements
17 cu = Trans1D(u,xa,xb); cv = Trans1D(v,xa,xb);
18 K = K + cv.*cu.*J.*Kref;
19 end

```

有了 K , 其他过程都可按照前面介绍的处理, 即装配并处理边界条件. 完整的 Bilinear.m 如下

```

1 function kk = Bilinear1D(Th,varargin)
2
3 if nargin==3
4     Trial = varargin{1}; Test = varargin{2};
5     Coef = num2cell(1:length(Trial));
6 elseif nargin==4
7     Coef = varargin{1}; Trial = varargin{2}; Test = varargin{3};
8 else
9     error('Check the input');
10 end
11
12 node = Th.node; elem = Th.elem;
13 N = size(node,1); nel = size(elem,1); Ndof = 2;
14 xa = node(elem(:,1)); xb = node(elem(:,2));
15
16 K = zeros(nel,2*Ndof); % straighten for easy assembly
17 J = Jacobian1D(xa,xb); % Jacobian on all elements
18 [r, w] = GaussQuad(4,0,1); % on [0,1]
19 xx = xa+(xb-xa)*r'; % quadrature points on all elements
20 for s = 1:length(Coef) % s-th component of bilinear form
21     u = Trial{s}; v = Test{s};
22     % Trial and Test functions
23     [u1,u2] = Refbase1D(u); [v1,v2] = Refbase1D(v);
24     % Stiffness matrix on reference interval [0,1]
25     cf = Coef{s};
26     if isa(cf,'double'), cf = @(x) cf+0*x; end
27     cf = cf(xx);
28     v1 = v1(r); v2 = v2(r); u1 = u1(r); u2 = u2(r);
29     ke = w.*[v1.*u1, v1.*u2, v2.*u1, v2.*u2];
30     Kref = cf*ke;

```

```

31 % Stiffness matrix on all elements
32 cu = Trans1D(u,xa,xb); cv = Trans1D(v,xa,xb);
33 K = K + cv.*cu.*J.*Kref;
34 end
35
36 % sparse indices
37 nnz = nel*Ndof^2;
38 ii = zeros(nnz,1); jj = zeros(nnz,1); ss = K(:);
39 id = 0;
40 for i = 1:Ndof
41     for j = 1:Ndof
42         ii(id+1:id+nel) = elem(:,i); % zi
43         jj(id+1:id+nel) = elem(:,j); % zj
44         id = id + nel;
45     end
46 end
47
48 % stiffness matrix
49 kk = sparse(ii,jj,ss,N,N);

```

1.5.3 程序整理

主程序如下

CODE 1.4. main_FEM1D_variational.m

```

1 clc;clear; close all;
2 tic;
3 % ----- Mesh and boundary conditions -----
4 a = 0; b = 1;
5 nel = 10; N = nel+1; % numbers of elements and nodes
6 node = linspace(a,b,nel+1)';
7 elem = zeros(nel,2); elem(:,1) = 1:N-1; elem(:,2) = 2:N;
8 Th.node = node; Th.elem = elem;
9
10 Neumann = []; Dirichlet = [1,N];
11 bdStruct = struct('Dirichlet', Dirichlet, 'Neumann', Neumann);
12
13 % ----- PDE -----
14 acoef = -1; bcoef = 2; ccoef = 1;
15 para = struct('acoef', acoef, 'bcoef', bcoef, 'ccoef', ccoef);

```

```

16 pde = pdedata1D(para);
17
18 % ----- Assemble the matrix -----
19 acoef1 = -acoef;
20 Coef = {acoef1, bcoef, ccoef};
21 Trial = {'u.dx', 'u.dx', 'u.val'};
22 Test = {'v.dx', 'v.val', 'v.val'};
23 kk = Bilinear1D(Th, Coef, Trial, Test);
24
25 % ----- Assemble the vector -----
26 ff = Linear1D(Th, pde.f);
27
28 % ----- Apply boundary conditions -----
29 u = Applyboundary1D(Th, kk, ff, acoef, pde, bdStruct);
30
31 % ----- error analysis -----
32 uexact = pde.uexact(node);
33 figure, plot(node, u, 'r-', node, uexact, 'k--', 'linewidth', 1);
34 xlabel('x'); ylabel('u');
35 legend('Numerical solution', 'Exact solution')
36 Err = u-uexact;
37 figure, plot(node, Err, 'linewidth', 1); legend('Absolute error');
38 toc

```

这里, PDE 相关信息单独放在 pdedata1D.m 中. 对系数是函数的情形, 我们给出了 pdedata1D_variable.m, 主程序的部分如下修改

```

1 % ----- PDE -----
2 acoef = @(x) 1+x; bcoef = @(x) 1+x; ccoef = 2;
3 para = struct('acoef', acoef, 'bcoef', bcoef, 'ccoeff', ccoef);
4 pde = pdedata1D_variable(para);
5
6 % ----- Assemble the matrix -----
7 acoef1 = @(x) -acoef(x);
8 Coef = {acoef1, bcoef, ccoef};
9 Trial = {'u.dx', 'u.dx', 'u.val'};
10 Test = {'v.dx', 'v.val', 'v.val'};

```

当某些系数为零时, 可以直接去掉相应的组份. 可以看到, 经过上面的处理, 一维问题的线性变分问题都能处理, 只要按照变分形式书写三元组 (Coef, Trial, Test) 即可.

边界条件的处理也封装成一个函数

```
1 function u = Applyboundary1D(Th,kk,ff,acoef,pde,bdStruct)
2
3 node = Th.node; N = length(node);
4 if isa(acoef,'double'), acoef = @(x) acoef+0*x; end
5
6 % ----- Neumann boundary conditions -----
7 Neumann = bdStruct.Neumann;
8 if ~isempty(Neumann)
9     bn = [-1;zeros(N-2,1);1]; % -1: left norm vector
10    bn(Neumann) = bn(Neumann)*pde.g_N(node(Neumann));
11    ff = ff - acoef(node(Neumann))*bn;
12 end
13
14 % ----- Dirichlet boundary conditions -----
15 Dirichlet = bdStruct.Dirichlet;
16 isBdNode = false(N,1); isBdNode(Dirichlet) = true;
17 bdNode = (isBdNode); freeNode = (~isBdNode);
18 u = zeros(N,1); u(bdNode) = pde.g_D(node(bdNode));
19 ff = ff - kk*u;
20
21 % ----- Solver -----
22 u(freeNode) = kk(freeNode,freeNode)\ff(freeNode);
```

第二章 网格的图示与标记

2.1 网格与解的图示

2.1.1 基本数据结构

我们采用 Chen Long 有限元工具箱 iFEM 中给出的数据结构, 用 `node` 表示节点坐标, `elem` 表示单元的连通性, 即单元顶点编号. 例如考虑下图中 L 形区域的一个简单剖分 (对一般的多角形剖分类似). 可参考网页说明:

<https://www.math.uci.edu/~chenlong/ifemdoc/mesh/meshbasicdoc.html>

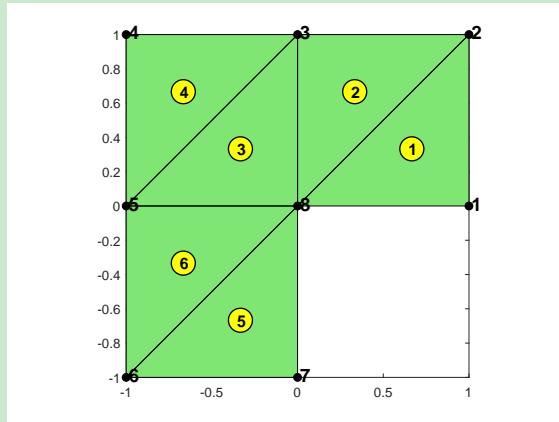


图 2.1. L 形区域的剖分

1. 数组 `node`: 节点坐标

在编程中我们需要每个节点的坐标, 用 `node` 记录, 它是两列的一个矩阵, 第一列表示各节点的横坐标, 第二列表示各节点的纵坐标, 行的索引对应节点标号. 图中给出的顶点坐标信息如下

8x2 double		
	1	2
1	1	0
2	1	1
3	0	1
4	-1	1
5	-1	0
6	-1	-1
7	0	-1
8	0	0

这里左侧的序号对应节点的整体编号.

2. 数组 elem: 连通性 (局部整体对应)

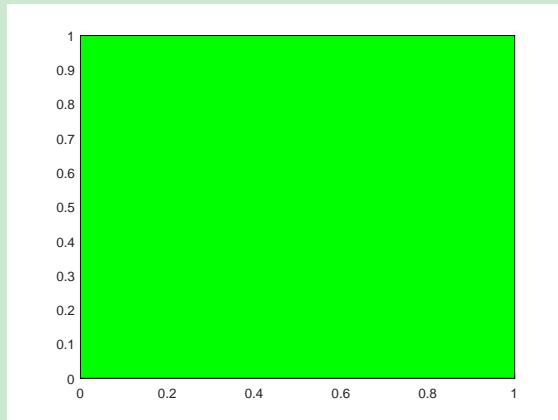
数组 elem 给出每个三角形的顶点编号, 它给出的是单元的连通性信息, 每行对应一个单元.

6x3 double			
	1	2	3
1	1	2	8
2	3	8	2
3	8	3	5
4	4	5	3
5	7	8	6
6	5	6	8

图中第一列表示所有三角形的第一个点的编号, 第二列表示第二个点的编号, 依此类推. 注意三角形顶点的顺序符合逆时针定向. elem 是有限元编程装配过程中的局部整体对应. 对多角形剖分, elem 为元胞数组, 每个元胞存储一个单元.

2.1.2 补片函数 patch

我们要画出每个单元, 对三角形单元 MATLAB 有专门的命令, 对多边形我们需要采用补片函数 patch. 实际上三角剖分采用的也是 patch, 为此本文只考虑 patch. 以下只考虑二维区域剖分的图示, 命名为 showmesh.m. 一个简单的例子如下图



可如下编程

```
node = [0 0; 1 0; 1 1; 0 1];
elem = [1 2 3 4];
patch('Faces',elem,'Vertices',node,'FaceColor','g')
```

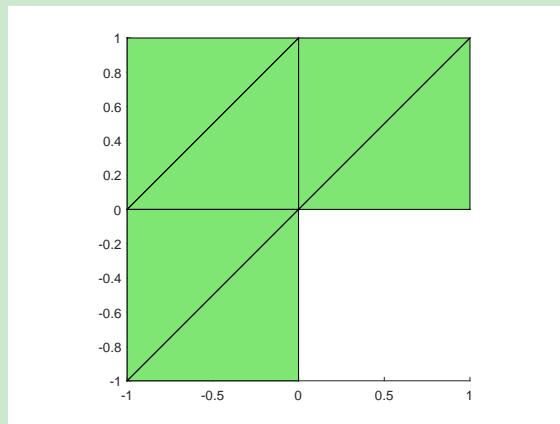
对多个相同类型的单元, 如下

```
1 function showmesh(node,elem)
2 h = patch('Faces',elem, 'Vertices', node);
```

```
3 set(h,'facecolor',[0.5 0.9 0.45],'edgecolor','k');
4 axis equal; axis tight;
```

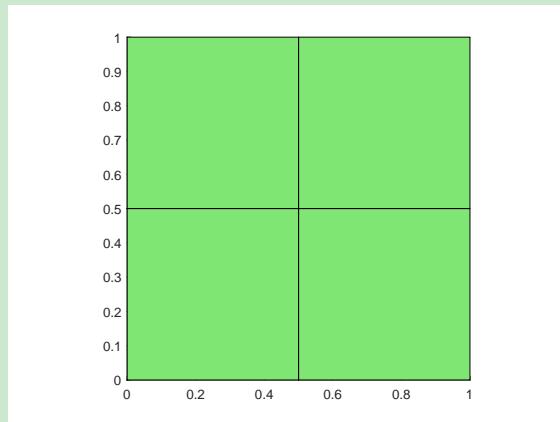
例 2.1 (三角剖分) 对前面的梯形区域可如下调用 showmesh 函数

```
1 node = [1,0; 1,1; 0,1; -1,1; -1,0; -1,-1; 0,-1; 0,0];
2 elem = [1,2,8; 3,8,2; 8,3,5; 4,5,3; 7,8,6; 5,6,8];
3 showmesh(node,elem);
```



例 2.2 (四边形剖分) 对矩形区域的四边形剖分可如下调用 showmesh 函数

```
1 [X,Y] = ndgrid(0:0.5:1,0:0.5:1);
2 node = [X(:), Y(:)];
3 elem = [1 2 5 4; 2 3 6 5; 4 5 8 7; 5 6 9 8];
4 showmesh(node,elem)
```



2.1.3 操作 cell 数组的 cellfun 函数

对含有不同多角形剖分的区域, 因每个单元顶点数不同, elem 一般以 cell 数组存储. 为了使用 patch 画图 (不用循环语句逐个), 我们需要将 elem 的每个 cell 填充成相同维数的向

量, 填充的值为 NaN, 它不会起作用. 我们先介绍 MATLAB 中操作 cell 数组的函数 cellfun. 例如, 考虑下面的例子

例 2.3 计算 cell 数组中元素的平均值和维数

```

1 C = {1:10, [2; 4; 6], []};
2 averages = cellfun(@mean, C)
3 [nrows, ncols] = cellfun(@size, C)
4
5 % 结果为 averages = 5.5000    4.0000      NaN
6 %      nrows = 1 3 0,      ncols = 10 1 0

```

cellfun 的直接输出规定为数值数组, 如果希望输出的可以是其他类型的元素, 那么需要指定 UniformOutput 为 false, 例如

例 2.4 对字符进行缩写

```

1 days = {'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday'};
2 abbrev = cellfun(@(x) x(1:3), days, 'UniformOutput', false)

```

正因为此时输出类型可以任意, MATLAB 默认仍保存为 cell 类型. 上面的结果为

```

abbrev =
1×5 cell array
{'Mon'}    {'Tue'}    {'Wed'}    {'Thu'}    {'Fri'}

```

2.1.4 showmesh 函数的建立

现在考虑下图所示的剖分

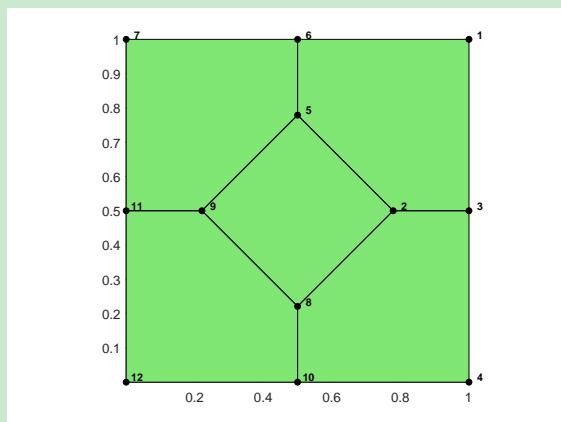


图 2.2. 多角形网格

相关的网格数据保持在 meshex1.mat 中. 程序如下

```

1 load('meshhex1.mat'); % node, elem
2
3 max_n_vertices = max(cellfun(@length, elem));
4 % function to pad the vacancies (横向拼接)
5 padding_func = @(vertex_ind) [vertex_ind, ...
6     NaN(1,max_n_vertices-length(vertex_ind))];
7 tpad = cellfun(padding_func, elem, 'UniformOutput', false);
8 tpad = vertcat(tpad{:});
9 h = patch('Faces', tpad, 'Vertices', node);
10 set(h,'facecolor',[0.5 0.9 0.45], 'edgecolor','k');
11 axis equal; axis tight;

```

最终给出的 showmesh 函数如下

CODE 2.1. showmesh.m (2D 网格画图)

```

1 function showmesh(node,elem)
2 %Showmesh displays a mesh in 2-D.
3
4 if ~iscell(elem)
5     h = patch('Faces', elem, 'Vertices', node);
6
7 else
8     max_n_vertices = max(cellfun(@length, elem));
9     padding_func = @(vertex_ind) [vertex_ind, ...
10         NaN(1,max_n_vertices-length(vertex_ind))]; % function to ...
11         pad the vacancies
12     tpad = cellfun(padding_func, elem, 'UniformOutput', false);
13     tpad = vertcat(tpad{:});
14     h = patch('Faces', tpad, 'Vertices', node);
15
16 set(h,'facecolor',[0.5 0.9 0.45], 'edgecolor','k');
17 axis equal; axis tight;

```

2.1.5 showsolution 函数的建立

节 4.3.5 已用到 showsolution 函数绘制解的网格图, 它的程序如下

```

1 function showsolution(node,elem,u)

```

```

2 %Showsolution displays the solution corresponding to a mesh given ...
3   by [node,elem] in 2-D.
4
5 data = [node,u];
6 patch('Faces', elem, ...
7       'Vertices', data, ...
8       'FaceColor', 'interp', ...
9       'CData', u / max(abs(u)) );
10 axis('square');
11 sh = 0.05;
12 xlim([min(node(:,1))-sh, max(node(:,1))+sh])
13 ylim([min(node(:,2))-sh, max(node(:,2))+sh])
14 xlabel('x'); ylabel('y'); zlabel('u');
15
16 view(3); grid on; % view(150,30);

```

我们来说明一下.

- patch 也可以画空间中的直面, 此时只要把 'Vertices' 处的数据换为三维的顶点坐标.
- 对解 u , 显然 $\text{data} = [\text{node}, \text{u}]$ 就是我们画图需要的三维点坐标.
- patch 后的

```
'FaceColor', 'interp', 'CData', u / max(abs(u))
```

是三维图形的颜色, 它根据 'CData' 数据进行插值获得 (不对颜色进行设置, 默认为黑色). 也可以改为二维的

```
set(h, 'facecolor', [0.5 0.9 0.45], 'edgecolor', 'k');
```

此时显示的只是一种颜色, 对解通常希望有颜色的变化.

- 需要注意的是, 即便是三维数据, 若不加最后的

```
view(3); grid on; %view(150,30);
```

给出的也是二维图 (投影, 即二维剖分图).

当然上面的程序也可改为适合多角形剖分的, 如下

CODE 2.2. showsolution.m

```

1 function showsolution(node,elem,u)
2 %Showsolution displays the solution corresponding to a mesh given ...
3 % by [node,elem] in 2-D.
4
5 data = [node,u];
6 if ~iscell(elem)
7     patch('Faces', elem, ...
8           'Vertices', data, ...
9           'FaceColor', 'interp', ...
10          'CData', u / max(abs(u)) );
11 else
12     max_n_vertices = max(cellfun(@length, elem));
13     padding_func = @(vertex_ind) [vertex_ind, ...
14                                 NaN(1,max_n_vertices-length(vertex_ind))]; % function to ...
15     % pad the vacancies
16     tpad = cellfun(padding_func, elem, 'UniformOutput', false);
17     tpad = vertcat(tpad{:});
18     patch('Faces', tpad, ...
19           'Vertices', data, ...
20           'FaceColor', 'interp', ...
21           'CData', u / max(abs(u)) );
22 end
23 axis('square');
24 sh = 0.05;
25 xlim([min(node(:,1)) - sh, max(node(:,1)) + sh])
26 ylim([min(node(:,2)) - sh, max(node(:,2)) + sh])
27 zlim([min(u) - sh, max(u) + sh])
28 xlabel('x'); ylabel('y'); zlabel('u');

view(3); grid on; % view(150,30);

```

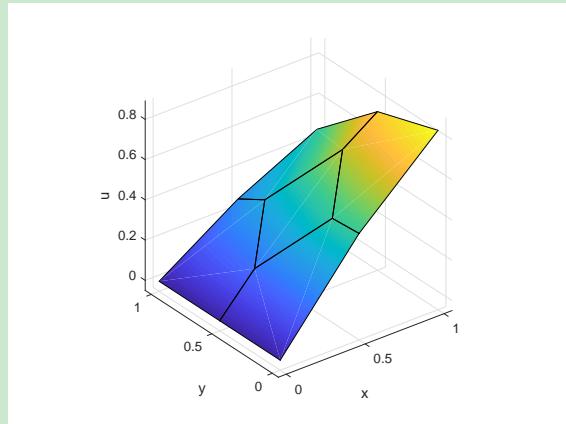
例 2.5 例如, 可如下画 $u(x,y) = \sin x \cos y$ 的图像

```

1 load('meshhex1.mat');
2 x = node(:,1); y = node(:,2); u = sin(x).*cos(y);
3 showsolution(node,elem,u);

```

结果如下



注 2.1 可以看到, showmesh 与 showsolution 唯一不同的地方就是添加

```
view(3); grid on; %view(150,30);
```

三维网格的单元是多面体, 我们一般也是逐个面画图, 此时可在 showmesh 下添加上面的语句. 修改后的 showmesh 如下 (画图时三维的 elem 要存储为面)

CODE 2.3. showmesh.m

```

1 function showmesh(node,elem)
2 %Showmesh displays a mesh in 2-D and 3-D.
3
4 if ~iscell(elem)
5     h = patch('Faces', elem, 'Vertices', node);
6 else
7     max_n_vertices = max(cellfun(@length, elem));
8     padding_func = @(vertex_ind) [vertex_ind, ...
9         NaN(1,max_n_vertices-length(vertex_ind))]; % function to ...
10    pad the vacancies
11    tpad = cellfun(padding_func, elem, 'UniformOutput', false);
12    tpad = vertcat(tpad{:});
13    h = patch('Faces', tpad, 'Vertices', node);
14 end
15 dim = size(node,2);
16 if dim==3
17     view(3); set(h,'FaceAlpha',0.4); % 透明度
18 end
19
20 set(h,'facecolor',[0.5 0.9 0.45], 'edgecolor','k');
21 axis equal; axis tight;

```

显然, 用该函数也可画解的图像

```
1 load('meshhex1.mat');
2 x = node(:,1); y = node(:,2); u = sin(x).*cos(y);
3 % show the solution by using showmesh
4 data = [node,u];
5 showmesh(data,elem);
```

结果一致, 只不过图像的颜色是单一的罢了. 为了方便, 我们单独建立了 showsolution 函数.

2.2 网格的标记

2.2.1 节点标记

可如下给出图 2.2 中的节点编号

```
1 load('meshhex1.mat');
2 showmesh(node,elem);
3 findnode(node);
```

函数文件如下

CODE 2.4. findnode.m

```
1 function findnode(node,range)
2 %Findnode highlights nodes in certain range.
3
4 hold on
5 dotColor = 'k.';
6 if nargin==1
7     range = (1:size(node,1))';
8 end
9 plot(node(range,1),node(range,2),dotColor, 'MarkerSize', 15);
10 shift = [0.015 0.015];
11 text(node(range,1)+shift(1),node(range,2)+shift(2),int2str(range), ...
12      'FontSize',8,'FontWeight','bold'); % show index number
13 hold off
```

注 2.2 当然也可改为适用于三维情形, 这里略, 见 GitHub 上传程序 (tool 文件夹内).

2.2.2 单元标记

现在标记单元. 我们需要给出单元的重心, 从而标记序号 (重心的计算说明略).

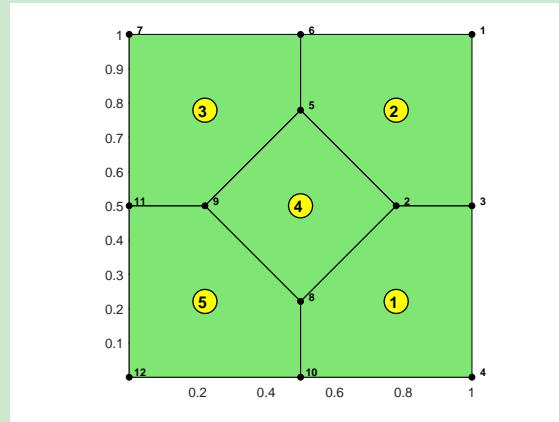


图 2.3. Polygonal mesh

主程序如下

```

1 load('meshhex1.mat');
2 showmesh(node,elem);
3 findnode(node);
4 findelem(node,elem);

```

标记单元的函数如下

CODE 2.5. findelem.m

```

1 function findelem(node,elem,range)
2 %Findelem highlights some elements
3
4 hold on
5
6 if nargin==2
7     range = (1:size(elem,1))';
8 end
9
10 center = zeros(length(range),2);
11 s = 1;
12 for iel = range(1):range(end)
13     if iscell(elem)
14         index = elem{iel};
15     else
16         index = elem(iel,:);
17     end
18     verts = node(index,:); verts1 = verts([2:end,1],:);
19     area_components = verts(:,1).*verts1(:,2)-verts1(:,1).*verts(:,2);

```

```

20     area = 0.5*abs(sum(area_components));
21     center(s,:) = ...
22         sum((verts+verts1).*repmat(area_components,1,2))/(6*area);
23 s = s+1;
24
25 plot(center(:,1),center(:,2),'o','LineWidth',1,'MarkerEdgeColor','k',...
26       'MarkerFaceColor','y','MarkerSize',18);
27 text(center(:,1)-0.02,center(:,2),int2str(range),'FontSize',12,...
28       'FontWeight','bold','Color','k');
29
30 hold off

```

注 2.3 这里用圆圈标记单元, 对不同的剖分, 圆圈内的数字不一定在合适的位置, 需要手动调整. 为了方便, 可直接用红色数字标记单元序号.

2.2.3 边的标记

Chen L 在如下网页

<https://www.math.uci.edu/~chenlong/ifemdoc/mesh/auxstructuredoc.html>

中给出了一些辅助网格数据结构 (三角剖分), 其中的 edge 就是记录每条边的顶点编号 (去除重复边). 以下设 NT 表示三角形单元的个数, NE 表示边的个数 (不重复). 我们简单说明一下那里的思路.

- 只要给出每条边两端的节点编号. 内部边在 elem 中会出现两次, 边界边只会出现一次, 我们可用 2 标记内部边, 1 标记边界边.
- 内部边在 elem 中会出现两次, 但它们是同一条边. 为了给定一致的标记, 我们规定每条边起点的顶点编号小于终点的顶点编号, 即 $\text{edge}(k, 1) < \text{edge}(k, 2)$.
- 规定三角形的第 i 条边对应第 i 个顶点 (不是必须的, 这个规定有利于网格二分程序的实现), 例如, 设第 1 个三角形顶点顺序为 [1,4,5], 那么边的顺序应是 4-5, 5-1, 1-4. 在 MATLAB 中, 有如下对应

所有单元的第 1 条边: `elem(:, [2,3]); % NT * 2`

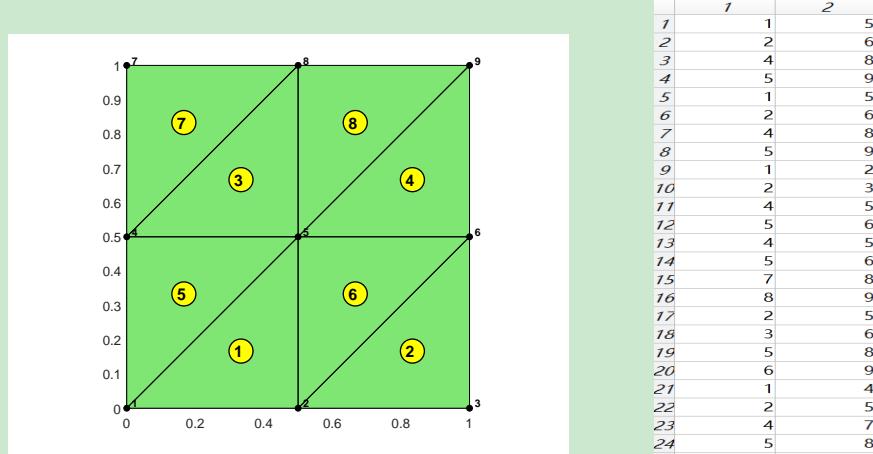
所有单元的第 2 条边: `elem(:, [3,1]); % NT * 2`

所有单元的第 3 条边: `elem(:, [1,2]); % NT * 2`

为了满足 $\text{edge}(k, 1) < \text{edge}(k, 2)$, 可对以上每个矩阵按行进行排列 (每行的两个元素进行比较). 在 MATLAB 中用 `sort(A, 2)` 实现. 把这些边逐行排在一起, 则所有的边 (包含重复) 为

```
totalEdge = sort([elem(:, [2, 3]); elem(:, [3, 1]); elem(:, [1, 2])], 2);
```

它是 $3NT * 2$ 的矩阵. `totalEdge` 见下面的右图.



- 在 MATLAB 中, `sparse` 有一个特殊的性质 (summation property), 当某个位置指标出现两次, 则相应的值会相加. 这样, 使用如下命令 (`sparse(i, j, s)`, 若 `s` 为固定常数, 直接写常数即可)

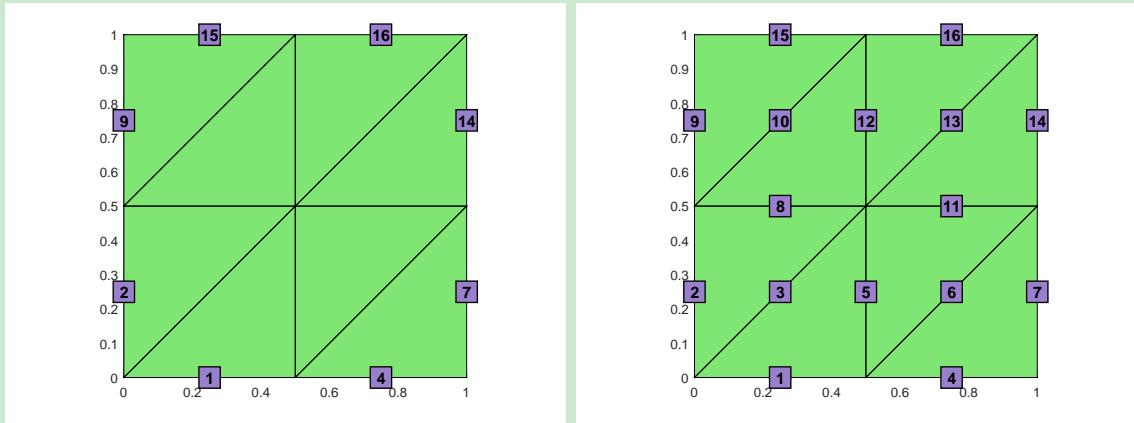
```
sparse(totalEdge(:, 1), totalEdge(:, 2), 1)
```

1-5 边对应的位置 (1,5) 的值就是 2, 而 1-2 对应的位置 (1,2) 为 1, 即重复边的都是 2, 不重复的为 1. 用 `find` 可找到所有非零元素的位置及相应的值 (非零元素只有 1 和 2, 对应边界边和内部边). 显然, `sparse` 命令产生的矩阵, 第一行对应起点 1 的边, 第二行对应起点 2 的边, 等等.

- 我们希望按下列方式排列边: 先找到所有起点为 1 的边, 再找所有起点为 2 的边, 等等. 由于 `find` 是按列找非零元素, 因此我们要把上一步的过程如下修改 (转置)

```
sparse(totalEdge(:, 2), totalEdge(:, 1), 1)
```

这样, 第一列对应的是起点为 1 的边, 第二列对应的是起点为 2 的边.



综上，我们可如下标记边界边或所有的边

```

1 % ----- edge -----
2 [node,elem] = squaremesh([0 1 0 1],0.5);
3 figure, % boundary edges
4 showmesh(node,elem);
5 bdInd = 1;
6 findedgeTr(node,elem,bdInd);
7 figure, % all edges
8 showmesh(node,elem);
9 findedgeTr(node,elem);

```

函数文件如下

```

1 function findedgeTr(node,elem,bdInd)
2 %FindedgeTr highlights edges for triangulation
3 % bdEdge = 1; % boundary edge;
4 % other cases: all edges
5
6 hold on
7 % ----- edge matrix -----
8 totalEdge = sort([elem(:,[2,3]); elem(:,[3,1]); elem(:,[1,2])],2);
9 [i,j,s] = find(sparse(totalEdge(:,2),totalEdge(:,1),1));
10 edge = [j,i];
11 % bdEdge = edge(s==1,:);
12
13 % ----- range -----
14 if nargin==2 || bdInd!=1
15     range = (1:size(edge,1))'; % all edges
16 else
17     range = find(s==1); % boundary edges

```

```

18 end
19
20 % ----- edge index -----
21 midEdge = (node(edge(range,1),:)+node(edge(range,2),:))/2;
22 plot(midEdge(:,1),midEdge(:,2),'s','LineWidth',1,'MarkerEdgeColor','k',...
23 'MarkerFaceColor',[0.6 0.5 0.8],'MarkerSize',20);
24 text(midEdge(:,1)-0.025,midEdge(:,2),int2str(range),...
25 'FontSize',12,'FontWeight','bold','Color','k');

```

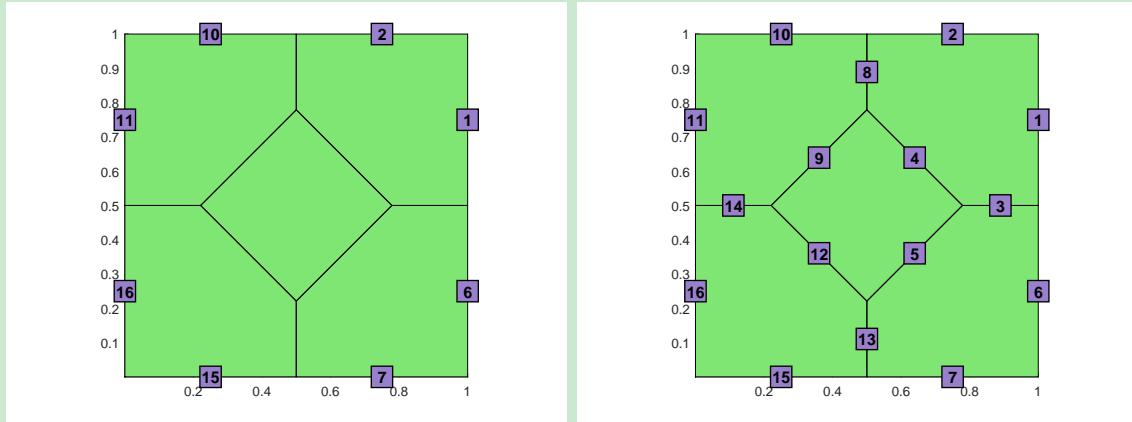
注意因为前面进行了转置, `edge = [j,i]`.

上面的思想也可用于多角形剖分, 只不过因边数不同, 要逐个单元存储每条边. 图 2.3 中第 1 个单元的顶点顺序为 [8,10,4,3,2], 我们按顺序 8-10, 10-4, 4-3, 3-2, 2-8 给出单元的边的标记. 所有边的起点就是 `elem` 中元素按列拉直给出的结果, 而终点就是对 [10,4,3,2,8] 这种循环的结果拉直. 可如下实现

```

1 % the starting points of edges
2 v0 = horzcat(elem{:})';
3
4 % the ending points of edges
5 shiftfun = @(verts) [verts(2:end),verts(1)];
6 T1 = cellfun(shiftfun, elem, 'UniformOutput', false);
7 v1 = horzcat(elem{:})';

```



其他过程与三角剖分一致. 如下运行

```

1 % ----- edge (polygonal meshes) -----
2 load('meshhex1.mat');
3 figure, % boundary edges
4 showmesh(node,elem);

```

```

5 bdInd = 1;
6 findedge(node, elem, bdInd)
7 figure, % all edges
8 showmesh(node, elem);
9 findedge(node, elem)

```

函数文件如下

CODE 2.6. fidedge.m

```

1 function fidedge(node, elem, bdInd)
2 %Finedge highlights edges
3 % bdEdge = 1; % boundary edge;
4 % other cases: all edges
5
6 hold on
7 % ----- edge matrix -----
8 if iscell(elem)
9     shiftfun = @(verts) [verts(2:end),verts(1)];
10    T1 = cellfun(shiftfun, elem, 'UniformOutput', false);
11    v0 = horzcat(elem{:})'; % the starting points of edges
12    v1 = horzcat(T1{:})'; % the ending points of edges
13    totalEdge = sort([v0,v1],2);
14 else
15    totalEdge = sort([elem(:,[2,3]); elem(:,[3,1]); elem(:,[1,2])],2);
16 end
17 [i,j,s] = find(sparse(totalEdge(:,2),totalEdge(:,1),1));
18 edge = [j,i];
19 % bdEdge = edge(s==1,:);
20
21 % ----- range -----
22 if nargin==2 || bdInd!=1
23    range = (1:size(edge,1))'; % all edges
24 else
25    range = find(s==1); % boundary edges
26 end
27
28 % ----- edge index -----
29 midEdge = (node(edge(range,1),:)+node(edge(range,2),:))/2;
30 plot(midEdge(:,1),midEdge(:,2),'s','LineWidth',1,'MarkerEdgeColor','k',...
31      'MarkerFaceColor',[0.6 0.5 0.8],'MarkerSize',20);

```

```
32 text(midEdge(:,1)-0.025,midEdge(:,2),int2str(range), ...
33     'FontSize',12,'FontWeight','bold','Color','k');
```

注 2.4 如果只是单纯生成 edge 矩阵, 那么也可如下

```
edge = unique(totalEdge,'rows');
```

unique 的速度要比前面给出的方式慢, 但后者方式可以用来生成对应单元的边界, 命名为 elem2edge, 它在计算中更重要.

第三章 辅助数据结构与几何量

网格中有许多数据在计算中很有用, 例如边的标记、单元的直径、面积等. 参考 iFEM 的相关内容, 见网页

<https://www.math.uci.edu/~chenlong/ifemdoc/mesh/auxstructuredoc.html>

本章针对一般的多角形剖分给出需要的数据结构与几何量.

3.1 辅助数据结构

我们的数据结构包括

表 3.1. 数据结构

node, elem	基本数据结构
elem2edge	边的自然序号 (单元存储)
edge	一维边的端点标记
bdEdge	边界边的端点标记
edge2elem	边的左右单元
neighbor	目标单元边的相邻单元

3.1.1 elem2edge 的生成

elem2edge 按单元记录每条边的自然序号, 这里同时会给出 edge, bdEdge.

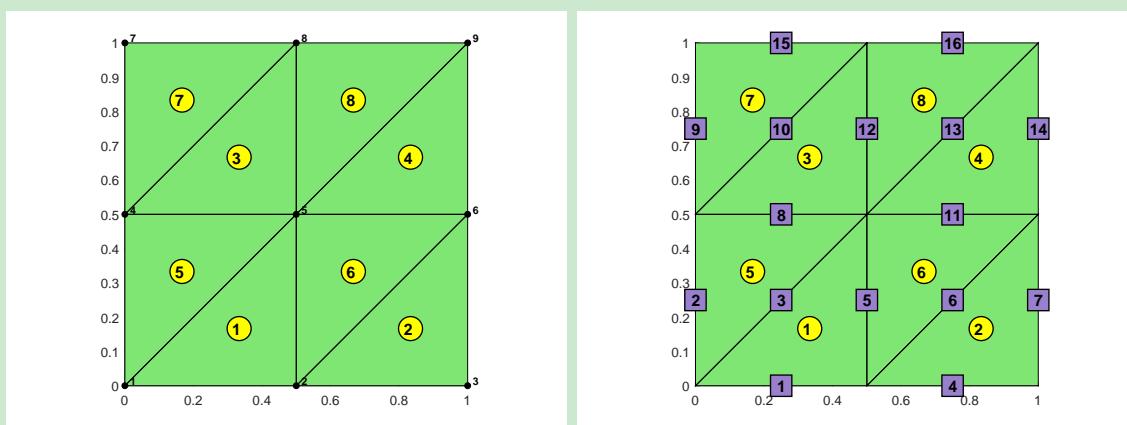


图 3.1. 三角剖分边的自然序号

考虑图 3.1 中给出的三角剖分, 我们说明一下 iFEM 中的思路.

- 根据前面的说明, 我们可给出含重复边的数组 `totalEdge` 见图 3.2(a).

	1	2
1	1	5
2	2	6
3	4	8
4	5	9
5	1	5
6	2	6
7	4	8
8	5	9
9	1	2
10	2	3
11	4	5
12	5	6
13	4	5
14	5	6
15	7	8
16	8	9
17	2	5
18	3	6
19	5	8
20	6	9
21	1	4
22	2	5
23	4	7
24	5	8

	1	2
1	1	2
2	1	4
3	1	5
4	2	3
5	2	5
6	2	6
7	3	6
8	4	5
9	4	7
10	4	8
11	5	6
12	5	8
13	5	9
14	6	9
15	7	8
16	8	9
17	9	10
18	10	11
19	11	12
20	12	13
21	13	14
22	14	15
23	15	16
24	16	17

	1
1	9
2	21
3	1
4	10
5	17
6	2
7	18
8	11
9	23
10	3
11	12
12	19
13	4
14	20
15	15
16	16
17	5
18	7
19	12
20	14
21	2
22	5
23	9
24	12

	1	3
1	2	6
3	10	13
4	13	3
5	3	6
6	6	10
7	10	13
8	13	1
9	1	11
10	4	8
11	8	11
12	11	8
13	8	14
14	11	11
15	15	15
16	16	16
17	5	7
18	7	12
19	12	14
20	14	2
21	2	22
22	5	5
23	9	9
24	12	12

图 3.2. elem2edge 图示

- 如下可去除重复的行, 即重复的边 (重复边一致化才能使用)

```
[edge, i1, totalJ] = unique(totalEdge, 'rows');
```

这里, `edge` 是 $NE \times 2$ 的矩阵, 对应边的集合, 注意 `unique` 会按第一列从小到大给出边 (相应地第二列也进行了排序), 见图 3.2(b).

`i1` 是 $NE \times 1$ 的数组, 它记录 `edge` 中的每条边在原来的 `totalEdge` 的位置 (重复的按第一次出现记录). 比如, 上面的 1-5 边, 第一次出现的序号是 1, 则 `i1` 第一个元素就是 1.

`totalJ` 记录的是 `totalEdge` 的每条边在 `edge` 中的自然序号. 比如, 1-5 在 `edge` 中是第 3 个, 则 `totalEdge` 的所有 1-5 边的序号为 3.

- 只要把 `totalJ` 恢复成三列即得所有三角形单元边的自然序号, 这是因为 `totalEdge` 排列的规则是: 前 NT 行对应所有单元的第 1 条边, 中间 NT 行对应第 2 条边, 最后 NT 行对应第 3 条边. 综上, 可如下获取 `elem2edge`.

```

1 % ----- elem2edge (triangulation) -----
2 [node, elem] = squaremesh([0 1 0 1], 0.5);
3 totalEdge = sort([elem(:, [2, 3]); elem(:, [3, 1]); elem(:, [1, 2])], 2);
4 [edge, i1, totalJ] = unique(totalEdge, 'rows');
5 NT = size(elem, 1);
6 elem2edge = reshape(totalJ, NT, 3);

```

结果如下

	1	2	3
1	3	1	5
2	6	4	7
3	10	8	12
4	13	11	14
5	3	8	2
6	6	11	5
7	10	15	9
8	13	16	12

上面的思路适用于多角形剖分, 只不过此时因边数不同, 要逐个单元存储每条边, 以保证对应 (三角形按局部边存储可快速恢复). 当我们获得 `totalJ` 后, 它与 `totalEdge` 的行对应, 从而可对应 `elem` 获得 `elem2edge`. 在 MATLAB 中可用 `mat2cell` 实现, 请参考相关说明. 我们把前面获取 `edge`, `bdEdge` 以及这里的 `elem2edge` 的过程放在一个 M 文件中

```

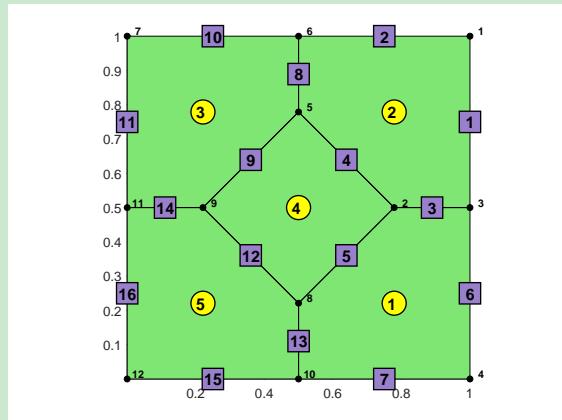
1 % % ----- elem2edge (triangulation) -----
2 % [node,elem] = squaremesh([0 1 0 1],0.5);
3 % showmesh(node,elem); findelem(node,elem); findnode(node);
4 % findedge(node,elem);
5
6 % ----- elem2edge (polygonal meshes) -----
7 load('meshhex1.mat');
8 showmesh(node,elem);
9 findnode(node); findelem(node,elem);
10 findedge(node,elem);
11
12 if iscell(elem)
13     % totalEdge
14     shiftfun = @(verts) [verts(2:end),verts(1)]; % or shiftfun = ...
15     %     @(verts) circshift(verts,-1);
16     T1 = cellfun(shiftfun, elem, 'UniformOutput', false);
17     v0 = horzcat(elem{:})'; % the starting points of edges
18     v1 = horzcat(T1{:})'; % the ending points of edges
19     totalEdge = sort([v0,v1],2);
20
21     % elem2edge
22     [~, ~, totalJ] = unique(totalEdge, 'rows');
23     elemLen = cellfun('length',elem); % length of each element
24     elem2edge = mat2cell(totalJ,elemLen,1);

```

```

24     elem2edge = cellfun(@transpose, elem2edge, 'UniformOutput', false);
25
26 else % Triangulation
27     totalEdge = sort([elem(:,[2,3]); elem(:,[3,1]); elem(:,[1,2])],2);
28     [~, ~, totalJ] = unique(totalEdge, 'rows');
29     NT = size(elem,1);
30     elem2edge = reshape(totalJ,NT,3);
31 end
32
33 % ----- edge, bdEdge -----
34 [i,j,s] = find(sparse(totalEdge(:,2),totalEdge(:,1),1));
35 edge = [j,i];
36 bdEdge = edge(s==1,:);

```



结果如下

5x1 cell
1
1 [13,7,6,3,5]
2 [3,1,2,8,4]
3 [14,9,8,10,11]
4 [12,5,4,9]
5 [15,13,12,14,16]

(a) elem2edge

16x2 double	
1	3
2	6
3	5
4	8
5	4
6	4
7	10
8	6
9	9
10	7
11	11
12	9
13	10
14	11
15	12
16	12

(b) edge

8x2 double	
1	3
2	6
3	4
4	10
5	7
6	11
7	12
8	12

(c) bdEdge

图 3.3. Auxiliary mesh data structure

3.1.2 edge2elem 的生成

对给定的一条边 e , 有时候希望知道包含它的单元有哪些. 对内部边, 就是哪两个单元以 e 为公共边. 为此, 我们定义矩阵 `edge2elem`, 维数为 $NE \times 2$, 其中 NE 是一维边的个数. 它的前两列分别存储相邻的三角形编号. 为了方便, 称第一列为左单元编号, 第二列为右单元编号. 注意, 对边界边我们规定两个编号一致.

	1	2	5
1	1		
2	2	6	
3	4	8	
4	5	9	
5	1	5	
6	2	6	
7	4	8	
8	5	9	
9	1	2	
10	2	3	
11	4	5	
12	5	6	
13	4	5	
14	5	6	
15	7	8	
16	8	9	
17	2	5	
18	3	6	
19	5	8	
20	6	9	
21	1	4	
22	2	5	
23	4	7	
24	5	8	

16x2 double		
1	1	2
2	1	4
3	1	5
4	2	3
5	2	5
6	2	6
7	3	6
8	4	5
9	4	7
10	4	8
11	5	6
12	5	8
13	6	7
14	6	9
15	7	10
16	7	13
17	8	1
18	8	4
19	9	10
20	9	13
21	10	11
22	10	14
23	11	15
24	11	12

16x1 double	
1	9
2	21
3	1
4	10
5	3
6	6
7	17
8	2
9	18
10	11
11	11
12	23
13	4
14	20
15	15
16	16
17	5
18	7
19	12
20	14
21	2
22	5
23	9
24	12

24x1 double	
1	3
2	6
3	10
4	13
5	3
6	6
7	10
8	13
9	1
10	4
11	8
12	11
13	8
14	11
15	15
16	16
17	5
18	7
19	12
20	14
21	2
22	5
23	9
24	12

(a) totalEdge

(b) edge

(c) i1

(d) totalJ

- `totalEdge` 记录了所有的重复边, 称第一次出现的重复边为左单元边, 第二次出现的重复边为右单元边. 根据前面的说明,

```
[~, i1, totalJ] = unique(totalEdge, 'rows');
```

执行上面语句给出的 `i1` 记录了左单元边.

- 类似地, 对 `totalEdge` 的逆序使用 `unique`:

```
[~, i2] = unique(totalEdge(end:-1:1,:), 'rows');
```

或

```
[~, i2] = unique(totalJ(end:-1:1), 'rows');
```

给出的 `i2` 记录了左单元边, 但现在的序号与原先的有差别. 以图中的例子为例, 此时 1 相当于原来的 24, 2 相当于 23, 依此类推. 它们的和总是 25, 即 `length(totalEdge)+1` (三角形为 $3 \times NT + 1$). 这样, 还原后的为

```
i2 = length(totalEdge)+1-i2;
```

- `totalJ` 或 `totalEdge` 并不是逐个单元存储的. 对三角剖分, 它是如下存储的

所有单元的第 1 条边: `elem(:, [2,3]); % NT * 2`

所有单元的第 2 条边: `elem(:, [3,1]); % NT * 2`

所有单元的第 3 条边: `elem(:, [1,2]); % NT * 2`

即, 前 NT 行与所有单元的第 1 条边对应, 中间的 NT 行与所有单元的第 2 条边对应, 最后的 NT 行与所有单元的第 3 条边对应. 设 totalJ 行的单元序号为 totalJelem, 则

```
totalJelem = repmat((1:NT)', 3, 1);
```

- 综上, 对三角形剖分, 有

```
edge2elem = totalJelem([i1, i2]);
```

- 对多角形剖分, 只要修改 totalJelem 即可. 对多角形情形, totalEdge 是按单元排列的, 只要按单元边数进行编号即可. 例如, 前面给出的例子, 每个单元的边数为

5x1 double	
1	
1	5
2	5
3	5
4	4
5	5

这里, 第 1 个单元有 5 条边, 第 2 个单元有 5 条边, 等等. 为此, totalJelem 的前 5 行都为 1 (对应单元 1), 接着的 5 行为 2, 等等. 如下给出

```
1 Num = num2cell((1:NT)');
2 Len = num2cell(cellLen);
3 totalJelem = cellfun(@(n1,n2) n1*ones(n2,1), Num, Len, ...
'UniformOutput', false);
4 totalJelem = vertcat(totalJelem{:});
```

综上, 可如下实现 edge2elem

```
1 % ----- edge2elem -----
2 if iscell(elem)
3     Num = num2cell((1:NT)'); Len = num2cell(cellLen);
4     totalJelem = cellfun(@(n1,n2) n1*ones(n2,1), Num, Len, ...
'UniformOutput', false);
5     totalJelem = vertcat(totalJelem{:});
6 else
7     totalJelem = repmat((1:NT)', 3, 1);
8 end
9 [~, i2] = unique(totalJ(end:-1:1), 'rows');
10 i2 = length(totalEdge)+1-i2;
11 edge2elem = totalJelem([i1, i2]);
```

多角形剖分结果如下

16x2 double	
1	2
2	2
3	1
4	2
5	1
6	1
7	1
8	2
9	3
10	3
11	3
12	4
13	1
14	3
15	5
16	5

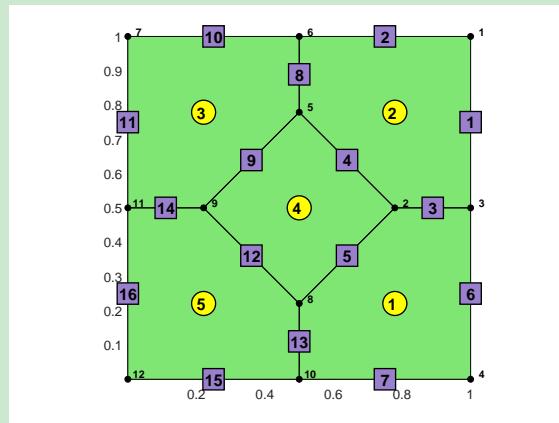


图 3.4. edge2elem

例如, 序号 12 的边连接的两个单元编号为 4 和 5.

注 3.1 totalEdge 是按单元顺序排列的, i_1 对应 e 第一次出现的单元, i_2 对应第二次出现的单元, 自然 edge2elem 的第一列单元序号小于或等于第二列单元序号.

3.1.3 neighbor 的生成

neighbor 是 $NT \times NE$ 的矩阵, 它的结构如下

neighbor 的结构		
i	j	$\text{neighbor}(i, j)$
K_i	e_j	相邻三角形

edge2elem(:, 1) 对应左单元 K_i , 行索引对应边 e_j , 相邻三角形是 edge2elem(:, 2).

edge2elem(:, 2) 对应右单元 K_i , 行索引对应边 e_j , 相邻三角形是 edge2elem(:, 1).

可用 sparse 实现 neighbor.

```

1 % ----- neighbor -----
2 NE = size(edge,1);
3 ii1 = edge2elem(:,1); jj1 = (1:NE)'; ss1 = edge2elem(:,2);
4 ii2 = edge2elem(:,2); jj2 = (1:NE)'; ss2 = edge2elem(:,1);
5 label = (ii2≠ss2);
6 ii2 = ii2(label); jj2 = jj2(label); ss2 = ss2(label);
7 ii = [ii1;ii2]; jj = [jj1;jj2]; ss = [ss1;ss2];
8 neighbor = sparse(ii,jj,ss,NT,NE);

```

注 3.2 本文给出的 neighbor 与 iFEM 不同, 那里是按单元给出每个顶点相对的单元序号, 这是由三角形的特殊性决定的.

3.2 网格相关的几何量

几何量包括

表 3.2. 几何量

Centroid	单元重心坐标
area	单元面积
diameter	单元直径

- 单元的重心如下计算

$$x_K = \frac{1}{6|K|} \sum_{i=0}^{N_v-1} (x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i),$$

$$y_K = \frac{1}{6|K|} \sum_{i=0}^{N_v-1} (y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i),$$

这里 N_v 是单元顶点个数.

- 单元面积

$$|K| = \frac{1}{2} \left| \sum_{i=0}^{N_v-1} x_i y_{i+1} - x_{i+1} y_i \right|.$$

- 单元直径就是所有顶点之间最长的距离, MATLAB 提供了 pdist 函数, 它计算各对行向量的相互距离.

以上几何量可如下获得

```

1 % ----- elemCentroid, area, diameter -----
2 Centroid = zeros(NT,2); area = zeros(NT,1); diameter = zeros(NT,1);
3 s = 1;
4 for iel = 1:NT
5     if iscell(elem)
6         index = elem{iel};
7     else
8         index = elem(iel,:);
9     end
10    verts = node(index, :); verts1 = verts([2:end,1],:);
11    area_components = verts(:,1).*verts1(:,2)-verts1(:,1).*verts(:,2);

```

```

12     ar = 0.5*abs(sum(area_components));
13     area(iel) = ar;
14     Centroid(s,:) = ...
15         sum((verts+verts1).*repmat(area_components,1,2))/(6*ar);
16     diameter(s) = max(pdist(verts));
17     s = s+1;
18 end

```

3.3 auxstructure 与 auxgeometry 函数

为了输出方便, 我们把所有的数据结构或几何量保存在结构体 aux 中. 考虑到数据结构在编程中不一定使用 (处理网格时用), 我们把数据结构与几何量分别用函数生成, 命名为 auxstructure.m 和 auxgeometry.m. 为了方便使用, 程序中把三角剖分按单元存储的数据转化为元胞数组. auxstructure.m 函数如下

CODE 3.1. auxstructure.m

```

1 function aux = auxstructure(node,elem)
2
3 NT = size(elem,1);
4 if iscell(elem)
5     % totalEdge
6     shiftfun = @(verts) [verts(2:end),verts(1)]; % or shiftfun = ...
6     @(verts) circshift(verts,-1);
7     T1 = cellfun(shiftfun, elem, 'UniformOutput', false);
8     v0 = horzcat(elem{:})'; % the starting points of edges
9     v1 = horzcat(T1{:})'; % the ending points of edges
10    totalEdge = sort([v0,v1],2);
11
12    % ----- elem2edge: elementwise edges -----
13    [~, i1, totalJ] = unique(totalEdge,'rows');
14    elemLen = cellfun('length',elem); % length of each elem
15    elem2edge = mat2cell(totalJ,elemLen,1);
16    elem2edge = cellfun(@transpose, elem2edge, 'UniformOutput', false);
17
18 else % Triangulation
19    totalEdge = sort([elem(:,[2,3]); elem(:,[3,1]); elem(:,[1,2])],2);
20    [~, i1, totalJ] = unique(totalEdge,'rows');
21    elem2edge = reshape(totalJ,NT,3);
22 end

```

```

23
24 % ----- edge, bdEdge -----
25 [i,j,s] = find(sparse(totalEdge(:,2),totalEdge(:,1),1));
26 edge = [j,i];
27 bdEdge = edge(s==1,:);
28
29 % ----- edge2elem -----
30 if iscell(elem)
31     Num = num2cell((1:NT)');      Len = num2cell(elemLen);
32     totalJelem = cellfun(@(n1,n2) n1*ones(n2,1), Num, Len, ...
33                           'UniformOutput', false);
34     totalJelem = vertcat(totalJelem{:});
35 else
36     totalJelem = repmat((1:NT)',3,1);
37 end
38 [~, i2] = unique(totalJ(end:-1:1), 'rows');
39 i2 = length(totalEdge)+1-i2;
40 edge2elem = totalJelem([i1,i2]);
41
42 % ----- neighbor -----
43 ii1 = edge2elem(:,1); jj1 = (1:NE)'; ss1 = edge2elem(:,2);
44 ii2 = edge2elem(:,2); jj2 = (1:NE)'; ss2 = edge2elem(:,1);
45 label = (ii2==ss2);
46 ii2 = ii2(label); jj2 = jj2(label); ss2 = ss2(label);
47 ii = [ii1;ii2]; jj = [jj1;jj2]; ss = [ss1;ss2];
48 neighbor = sparse(ii,jj,ss,NT,NE);
49
50
51 if ~iscell(elem) % transform to cell
52     elem = mat2cell(elem,ones(NT,1),3);
53     elem2edge = mat2cell(elem2edge,ones(NT,1),3);
54 end
55
56 aux.node = node; aux.elem = elem;
57 aux.elem2edge = elem2edge;
58 aux.edge = edge; aux.bdEdge = bdEdge;
59 aux.edge2elem = edge2elem;
60 aux.neighbor = neighbor;

```

auxgeometry.m 函数如下

CODE 3.2. auxgeometry.m

```
1 function aux = auxgeometry(node, elem)
2
3 % ----- elemCentroid, area, diameter -----
4 NT = size(elem,1);
5 Centroid = zeros(NT,2); area = zeros(NT,1); diameter = zeros(NT,1);
6 s = 1;
7 for iel = 1:NT
8     if iscell(elem)
9         index = elem{iel};
10    else
11        index = elem(iel,:);
12    end
13    verts = node(index, :); verts1 = verts([2:end,1],:);
14    area_components = verts(:,1).*verts1(:,2)-verts1(:,1).*verts(:,2);
15    ar = 0.5*abs(sum(area_components));
16    area(iel) = ar;
17    Centroid(s,:) = ...
18        sum((verts+verts1).*repmat(area_components,1,2))/(6*ar);
19    diameter(s) = max(pdist(verts));
20    s = s+1;
21
22 if ~iscell(elem) % transform to cell
23     elem = mat2cell(elem, ones(NT,1), 3);
24 end
25
26 aux.node = node; aux.elem = elem;
27 aux.Centroid = Centroid;
28 aux.area = area;
29 aux.diameter = diameter;
```

3.4 边界设置

假设网格的边界只有 Dirichlet 与 Neumann 两种类型, 前者用 `eD` 存储 Dirichlet 节点的编号, 后者用 `elemN` 存储 Neumann 边界的起点和终点编号 (即一维问题的连通性信息). 为了方便, 有时候需要 Dirichlet 边的信息, 为此我们用 `elemD` 存储 Dirichlet 边界的起点和终

点编号.

3.4.1 边界边的定向

辅助数据结构中曾给出了边界边 `bdEdge`, 但它的定向不再是逆时针, 因为我们规定 $\text{edge}(k, 1) < \text{edge}(k, 2)$. Neumann 边界条件中会遇到 $\partial_n u$, 这就需要我们恢复边界边的定向以确定外法向量(边的旋转获得).

给定 `totalEdge`, 即所有单元的边(含重复且无定向), 我们有两种方式获得边(第一种可获得边界边, 第二种只获得所有边):

- 一是累计重复的次数(1是边界, 2是内部)

```
1 [i,j,s] = find(sparse(totalEdge(:,2),totalEdge(:,1),1));
2 edge = [j,i];
3 bdEdge = edge(s==1,:);
```

- 二是直接去掉重复的边

```
1 [edge, i1, ~] = unique(totalEdge, 'rows');
```

这里, `i1` 记录的是 `edge` 在重复边 `totalEdge` 中的位置.

显然, `i1(s==1)` 给出的是边界边 `bdEdge` 在 `totalEdge` 中的位置. `totalEdge` 的原来定向是知道的, 由此就可确定边界边的定向, 程序如下

```
1 [node, elem] = squaremesh([0 1 0 1], 0.5);
2 NT = size(elem, 1);
3 % totalEdge
4 if iscell(elem)
5     shiftfun = @(verts) [verts(2:end), verts(1)];
6     T1 = cellfun(shiftfun, elem, 'UniformOutput', false);
7     v0 = horzcat(elem{:})'; % the starting points of edges
8     v1 = horzcat(T1{:})'; % the ending points of edges
9     allEdge = [v0, v1];
10    totalEdge = sort(allEdge, 2);
11 else
12     allEdge = [elem(:, [2, 3]); elem(:, [3, 1]); elem(:, [1, 2])];
13     totalEdge = sort(allEdge, 2);
14 end
15 [~, ~, s] = find(sparse(totalEdge(:, 2), totalEdge(:, 1), 1));
16 [edge, i1, ~] = unique(totalEdge, 'rows');
```

```
17 bdEdge = allEdge(i1(s==1), :); % counterclockwise
```

注 3.3 边界边 `bdEdge` 在一维边集合 `edge` 中的自然序号为

```
bdIndex = find(s==1);
```

3.4.2 边界的设置

下面说明如何实现 `eD`, `elemD` 和 `elemN`. 以下图为例

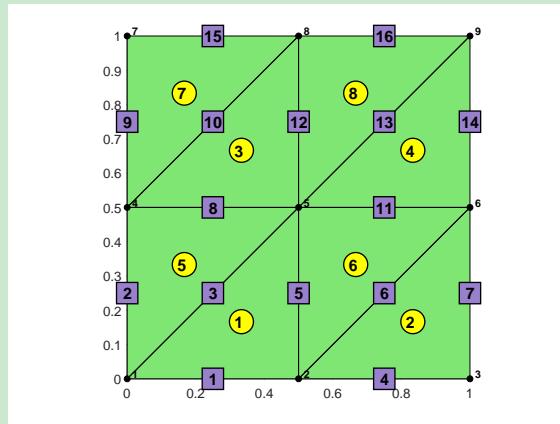


图 3.5. 边的自然序号

- 边界边的序号顺序为 $1, 2, 4, 7, 9, 14, 15, 16$. 定向的 `bdEdge` 给出的是这些边的起点与终点编号, 只要按索引对应即可.
- 边界我们用函数确定, 例如矩形 $[0,1]^2$ 的右边界为满足 $x = 1$ 的线段组成. 只需要判断 `bdEdge` 对应的边的中点在不在该线段上. 如下

```
1 bdFun = 'x==1';
2 nodebdEdge = (node(bdEdge(:,1),:) + node(bdEdge(:,2),:))/2;
3 x = nodebdEdge(:,1); y = nodebdEdge(:,2);
4 id = eval(bdFun);
```

这里, `eval` 将字符串视为语句并运行. 现在给定了若干个 `x`, 执行 `eval(bdFun)` 就会判断哪些 `x` 满足条件, 返回的是逻辑数组 `id = [0 0 0 1 0 1 0 0]`, 即索引中的第 4,6 条边在右边界上.

- 这样, 我们就可抽取出需要的边 `bdEdge(id,:)`. 需要注意的是, `node` 在边界上不一定精确为 1, 通常将上面的 `bdFun` 修改为

```
bdFun = 'abs(x-1)<1e-4';
```

- Neumann 边界通常比 Dirichlet 边界少, 为此在建函数的时候, 输入的字符串默认认为是 Neumann 边界的, 其他的都是 Dirichlet. 另外, 当没有边界字符串的时候, 规定所有边都是 Dirichlet 边.

根据上面的讨论, 我们可以给出函数 setboundary.m

CODE 3.3. setboundary.m

```

1 function bdStruct= setboundary(node, elem, varargin)
2 % varargin: string for Neumann boundary
3
4 % ----- totalEdge -----
5 if iscell(elem)
6     shiftfun = @(verts) [verts(2:end),verts(1)]; % or shiftfun = ...
7     @(verts) circshift(verts,-1);
8     T1 = cellfun(shiftfun, elem, 'UniformOutput', false);
9     v0 = horzcat(elem{:})'; % the starting points of edges
10    v1 = horzcat(T1{:})'; % the ending points of edges
11    allEdge = [v0,v1];
12 else % Triangulation
13     allEdge = [elem(:,[2,3]); elem(:,[3,1]); elem(:,[1,2])];
14 end
15 totalEdge = sort(allEdge,2);
16
17 % ----- counterclockwise bdEdge -----
18 [~,~,s] = find(sparse(totalEdge(:,2),totalEdge(:,1),1));
19 i1 = unique(totalEdge, 'rows');
20 bdEdge = allEdge(i1(s==1),:);
21
22 % ----- set boundary -----
23 nE = size(bdEdge,1);
24 % initial as Dirichlet (true for Dirichlet, false for Neumann)
25 bdFlag = true(nE,1);
26 nodebdEdge = (node(bdEdge(:,1),:) + node(bdEdge(:,2),:))/2;
27 x = nodebdEdge(:,1); y = nodebdEdge(:,2);
28 nvar = length(varargin); % 1 * size(varargin,2)
29 % note that length(varargin) = 1 for bdNeumann = [] or ''
30 if (nargin==2) || (~isempty(varargin{1}))
31     for i = 1:nvar
32         bdNeumann = varargin{i};
33         id = eval(bdNeumann);

```

```
33     bdFlag(id) = false;
34 end
35 end
36 bdStruct.eD = unique(bdEdge(bdFlag,:));
37 bdStruct.elemN = bdEdge(~bdFlag,:);
```

这里, `ed` 和 `elemN` 保存在结构体 `bdStruct` 中. 例如,

1. 以下命令给出的边界全是 Dirichlet 边界:

```
bdStruct = setboundary(node, elem);
bdStruct = setboundary(node, elem, []);
bdStruct = setboundary(node, elem, '');
```

2. `bdStruct = setboundary(node, elem, 'x==1')` 将右边界设为 Neumann 边, 其他为 Dirichlet 边.

3. `bdStruct = setboundary(node, elem, '(x==1)|(y==1)')` 将右边界与上边界设为 Neumann 边.

注 3.4 以下两种写法等价

```
bdStruct = setboundary(node, elem, '(x==1)|(y==1)');
bdStruct = setboundary(node, elem, 'x==1', 'y==1');
```

第四章 二维问题的有限元方法

从本章开始, 我们把主要精力放在二维问题上. 我们将会发现, 一维问题的处理策略, 可以容易地平移到二维问题上, 这也是本章需要实现的目标.

有限元编程可以归结为三步：

- 对区域进行剖分, 存储需要的网格数据;
 - 计算单元刚度矩阵和载荷向量, 并装配;
 - 计算代数方程组.

本章只考慮前兩步.

4.1 问题描述与网格数据

为了简单, 考虑 Poisson 问题

$$\begin{cases} -\Delta u = f & \text{in } \Omega, \\ u = g_D & \text{on } \Gamma_D, \\ \frac{\partial u}{\partial n} = g_N & \text{on } \Gamma_N. \end{cases} \quad (4.1)$$

计算中取精确解为

$$u(x, y) = y^2 \sin(\pi x).$$

区域及剖分如下图所示, 且假设右边界是 Neumann 边界条件, 其他为 Dirichlet 边界条件.

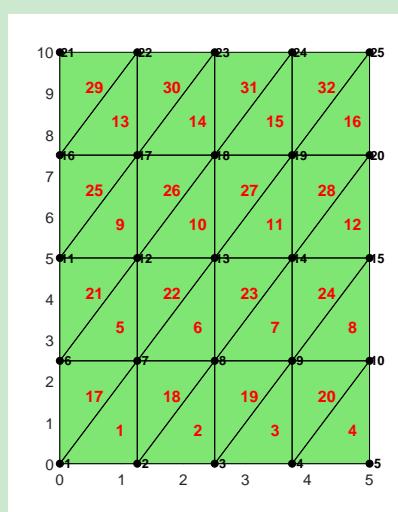


图 4.1. 矩形区域的标准三角剖分 (红色数字为单元编号)

不考虑边界条件, (4.1) 对应的变分形式为

$$\int_{\Omega} \nabla u \cdot \nabla v \, dx dy - \int_{\partial\Omega} v \frac{\partial u}{\partial n} \, dx dy = \int_{\Omega} f(x, y) \, dx dy.$$

如同一维问题, 我们最后考虑边界条件和边界积分

$$-\int_{\partial\Omega} v \frac{\partial u}{\partial n} \, dx dy.$$

如图 4.1, 矩形区域用标准三角形进行剖分, 红色数字表示单元编号, 黑色数字表示节点编号. 我们需要给出节点坐标、连通性以及局部整体编号关系等信息, 下面逐一考虑.

节点坐标

在编程中我们需要每个节点的坐标, 用 `node` 记录, 它是两列的一个矩阵, 第一列表示各节点的横坐标, 第二列表示各节点的纵坐标, 行的索引号对应节点的整体编号.

连通性

我们还要给出每个单元连通的顶点, 这里用 `elem` 表示 (每行对应一个单元), 如第 1 个单元连通的节点是 1, 2, 7, 有

```
1 elem(1,1)=1;      elem(1,2)=2;      elem(1,3)=7; % 第 1 个单元
```

局部整体对应

对每个单元三角形, 局部编号都是 1, 2, 3 (按逆时针方向). 如同一维问题, 在指标编程法中, 我们用 `index` 定位相应的整体编号. 例如, 对单元 ①, 可给出如下的局部整体对应

$$\{1, 2, 3\} \text{ (local)} \quad \rightarrow \quad \{1, 2, 7\} \text{ (global)},$$

这样对该单元有

```
index(1)= 1; index(2)= 2; index(3)= 7;
```

在连通性中我们已经获得了每个单元的顶点标号 (逆时针), 对单元 `iel`, 有

```
index = elem(iel,:);
```

剖分图 4.1 的 `node`, `elem` 可用如下函数生成 (参考 iFEM)

CODE 4.1. squaremesh.m

```
1 function [node,elem] = squaremesh(square,h1,h2)
2 %% SQUAREMESH uniform mesh of a square
3 %
```

```

4 % square = [a1,b1,a2,b2] for rectangle [a1,b1]*[a2,b2]
5 %
6 if nargin == 2, h2 = h1; end
7
8 % ----- Generate nodes -----
9 a1 = square(1); b1 = square(2); a2 = square(3); b2 = square(4);
10 [x,y] = ndgrid(a1:h1:b1,a2:h2:b2);
11 node = [x(:),y(:)];
12
13 % ----- Generate elements -----
14 nx = size(x,1); ny = size(y,2); % number of columns and rows
15
16 % 4 k+nx --- k+1+nx 3
17 % | |
18 % 1 k --- k+1 2
19
20 % indices of k
21 N = size(node,1);
22 k = (1:N-nx)'; cut = nx*(1:ny-1); k(cut) = [];
23
24 elem = [k+1 k+1+nx k; k+nx k k+1+nx];

```

注 4.1 有了 `node` 和 `elem`, 我们就可以画出剖分图 4.1 (参考 iFEM):

```
showmesh(node,elem); findnode(node); findelem(node,elem);
```

下一章再解释这些绘图函数的编写.

可通过如下方式获取 Neumann 和 Dirichlet 边界.

```

1 a1 = 0; b1 = 1; a2 = 0; b2 = 1; Nx = 4; Ny = 4;
2 square = [a1 b1 a2 b2]; h1 = (b1-a1)/Nx; h2 = (b2-a2)/Ny;
3
4 eps = 1e-4;
5 % ----- Set Dirichlet boundaries -----
6 idL = find(abs(node(:,1)-a1)≤eps); % left
7 idD = find(abs(node(:,2)-a2)≤eps); % down
8 idU = find(abs(node(:,2)-b2)≤eps); % upper
9 id = [idL;idD;idU];
10 eD = unique(id); % node id of Dirichlet boundaries
11
12 % ----- Set Neumann boundaries -----

```

```

13 idL = find(abs(node(:,1)-b1)≤eps); % right
14 yn = node(idL,2);
15 [~,m] = sort(yn); % sort in ascending order
16 id = idL(m);
17 % Neumann elements
18 nel = length(id)-1; % number of Neumann boundaries
19 elemN = zeros(nel,2);
20 % id of starting and ending points
21 elemN(:,1) = id(1:end-1); elemN(:,2) = id(2:end);

```

这里,

- eD 记录的是不重复的 Dirichlet 边界节点编号.
- elemN 记录的是 Neumann 边界单元, 即小区间的起点和终点编号.
- Neumann 边界之所以按一维的 elem 记录, 是因为 Neumann 条件可视为一维的装配问题, 后面将会看到.
- 下一章将介绍网格剖分, 在那里再给出一般性的处理.

注 4.2 我们把以上获取区域剖分以及边界条件的处理过程编写为函数文件, 用法如下

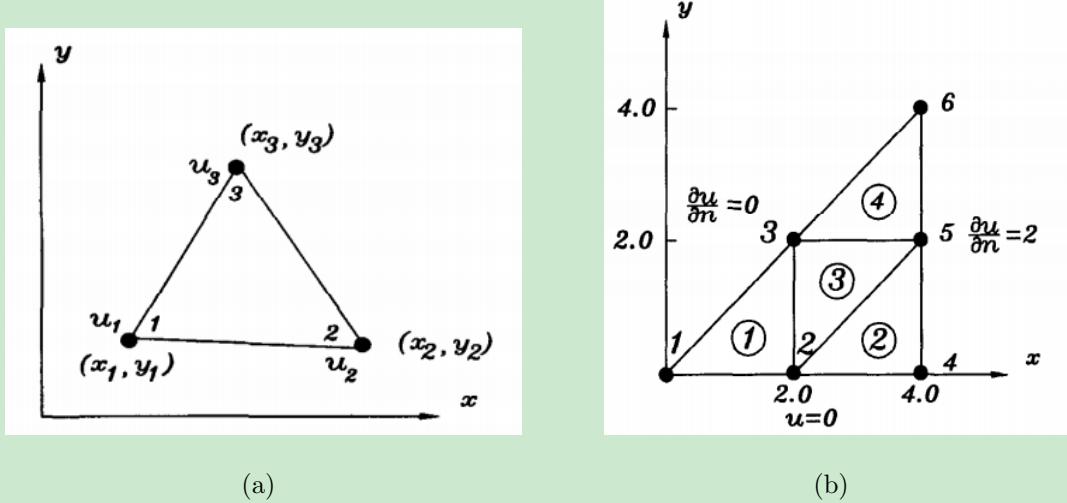
```
[node, elem, eD, elemN] = meshLap();
```

参数在该函数内部手动调整.

4.2 二维问题的装配

4.2.1 指标装配法

节 1.2 给出了一维问题刚度矩阵和载荷向量的装配算法, 其实对二维以及更高维问题, 该算法同样适用. 下面用一个例子来说明.



(a)

(b)

图 4.2. 单元剖分 (右图的圈表示单元编号)

以图 4.2 (b) 的剖分为例说明. 不考虑边界项, 变分形式的左端为

$$a(u, v) = \int_{\Omega} (u_x v_x + u_y v_y) dx dy.$$

为了方便, 考虑线性 Lagrange 有限元. 如图 4.2 (a), 单元上的线性插值为

$$u = \phi_1(x, y)u_1 + \phi_2(x, y)u_2 + \phi_3(x, y)u_3,$$

其中

$$\begin{aligned}\phi_1 &= \frac{1}{2S} [(x_2y_3 - x_3y_2) + (y_2 - y_3)x + (x_3 - x_2)y], \\ \phi_2 &= \frac{1}{2S} [(x_3y_1 - x_1y_3) + (y_3 - y_1)x + (x_1 - x_3)y], \\ \phi_3 &= \frac{1}{2S} [(x_1y_2 - x_2y_1) + (y_1 - y_2)x + (x_2 - x_1)y],\end{aligned}$$

满足

$$S = \frac{1}{2} \det \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix}, \quad \phi_i(z_j) = \delta_{ij}, \quad \phi_1 + \phi_2 + \phi_3 = 1.$$

注意到线性基函数求导后为常数, 对 $u = \phi_1u_1 + \phi_2u_2 + \phi_3u_3$, 逐个代入 $v = \phi_j$, $j = 1, 2, 3$ 易得单元刚度矩阵为

$$[K^e] = \begin{bmatrix} k_{11} & k_{12} & k_{13} \\ k_{21} & k_{22} & k_{23} \\ k_{31} & k_{32} & k_{33} \end{bmatrix},$$

这里

$$k_{11} = \frac{1}{4S} [(x_3 - x_2)^2 + (y_2 - y_3)^2],$$

$$k_{12} = k_{21} = \frac{1}{4S} [(x_3 - x_2)(x_1 - x_3) + (y_2 - y_3)(y_3 - y_1)],$$

$$k_{13} = k_{31} = \frac{1}{4S} [(x_3 - x_2)(x_2 - x_1) + (y_2 - y_3)(y_1 - y_2)],$$

$$k_{22} = \frac{1}{4S} [(x_1 - x_3)^2 + (y_3 - y_1)^2],$$

$$k_{23} = k_{32} = \frac{1}{4S} [(x_1 - x_3)(x_2 - x_1) + (y_3 - y_1)(y_1 - y_2)],$$

$$k_{33} = \frac{1}{4S} [(x_2 - x_1)^2 + (y_1 - y_2)^2].$$

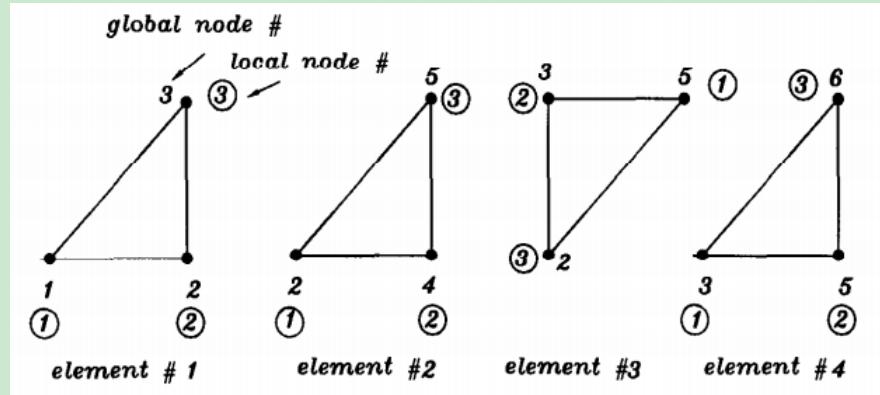


图 4.3. 局部与整体编号 (圈是局部标号)

若图 4.2 (b) 中的四个单元三角形都按图 4.3 进行局部编号 (四个单元的局部编号用带圈数字表示), 则每个单元的刚度矩阵都是

$$[K^e] = \begin{bmatrix} 0.5 & -0.5 & 0.0 \\ -0.5 & 1.0 & -0.5 \\ 0.0 & -0.5 & 0.5 \end{bmatrix}.$$

按照整体排序, 我们有 (原来在哪行, 还是应该在那行)

1. *element #1*

$$\begin{bmatrix} 0.5 & -0.5 & 0.0 \\ -0.5 & 1.0 & -0.5 \\ 0.0 & -0.5 & 0.5 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} \rightarrow \begin{bmatrix} 0.5 & -0.5 & 0.0 & 0 & 0 & 0 \\ -0.5 & 1.0 & -0.5 & 0 & 0 & 0 \\ 0.0 & -0.5 & 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \end{bmatrix}$$

2. element #2

$$\begin{bmatrix} 0.5 & -0.5 & 0.0 \\ -0.5 & 1.0 & -0.5 \\ 0.0 & -0.5 & 0.5 \end{bmatrix} \begin{bmatrix} u_2 \\ u_4 \\ u_5 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & -0.5 & 0.0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -0.5 & 0 & 1.0 & -0.5 & 0 \\ 0 & 0.0 & 0 & -0.5 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \end{bmatrix}$$

3. element #3

$$\begin{bmatrix} 0.5 & -0.5 & 0.0 \\ -0.5 & 1.0 & -0.5 \\ 0.0 & -0.5 & 0.5 \end{bmatrix} \begin{bmatrix} u_5 \\ u_3 \\ u_2 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.5 & -0.5 & 0 & 0.0 & 0 \\ 0 & -0.5 & 1.0 & 0 & -0.5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0.0 & -0.5 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \end{bmatrix}$$

4. element #4

$$\begin{bmatrix} 0.5 & -0.5 & 0.0 \\ -0.5 & 1.0 & -0.5 \\ 0.0 & -0.5 & 0.5 \end{bmatrix} \begin{bmatrix} u_3 \\ u_5 \\ u_6 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.5 & 0 & -0.5 & 0.0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -0.5 & 0 & 1.0 & -0.5 \\ 0 & 0 & 0.0 & 0 & -0.5 & 0.5 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \\ u_6 \end{bmatrix}$$

相加后我们就把单元刚度矩阵装配成整体刚度矩阵.

可以看到上面的处理与一维问题没有什么区别, 只是扩展时变量个数更多罢了. 显然只要给出局部编号与整体编号的关系向量 `index`, 就可按一维的算法合成整体刚度矩阵及载荷向量.

装配程序

指标装配程序如下

CODE 4.2. Get_kk_ff.m

```
1 function [kk,ff] = Get_kk_ff(node,elem)
```

```

2
3 N = size(node,1); NT = size(elem,1); % number of nodes and elements
4
5 kk = zeros(N,N); ff = zeros(N,1);
6 for iel = 1:NT
7     % local --> global
8     index = elem(iel,:);
9
10    % element matrix and vector
11    x1 = node(index(1),1); y1 = node(index(1),2); % coordinates of ...
12        element #iel
13    x2 = node(index(2),1); y2 = node(index(2),2);
14    x3 = node(index(3),1); y3 = node(index(3),2);
15
16    ke = feMatrix(x1,y1,x2,y2,x3,y3); % compute element matrix
17    fe = feVector(x1,y1,x2,y2,x3,y3); % compute element vector
18
19    % assemble
20    kk(index,index) = kk(index,index)+ke;
21    ff(index) = ff(index)+fe;
22 end
23 kk = sparse(kk);

```

这里单元刚度矩阵和单元载荷向量的程序如下

计算单元刚度矩阵 \mathbf{ke} 的函数文件

CODE 4.3. feMatrix.m

```

1 function ke = feMatrix(x1,y1,x2,y2,x3,y3)
2 % feMatrix: element matrix for two-dimensional Laplace's equation
3
4 Am = [1 x1 y1;1 x2 y2;1 x3 y3];
5 A = 1/2*det(Am); % area of the triangle
6 ke(1,1) = (x3-x2)^2+(y2-y3)^2;
7 ke(1,2) = (x3-x2)*(x1-x3)+(y2-y3)*(y3-y1);
8 ke(1,3) = (x3-x2)*(x2-x1)+(y2-y3)*(y1-y2);
9 ke(2,1) = ke(1,2);
10 ke(2,2) = (x1-x3)^2+(y3-y1)^2;
11 ke(2,3) = (x1-x3)*(x2-x1)+(y3-y1)*(y1-y2);

```

```

12 ke(3,1) = ke(1,3);
13 ke(3,2) = ke(2,3);
14 ke(3,3) = (x2-x1)^2+(y1-y2)^2;
15 ke = ke./(4*A);

```

计算单元载荷向量的函数文件

类似一维问题, 这里用中心格式近似积分

$$[F^e] = \int_K \begin{bmatrix} f\phi_1 \\ f\phi_2 \\ f\phi_3 \end{bmatrix} dx dy \approx \begin{bmatrix} f\lambda_1 \\ f\lambda_2 \\ f\lambda_3 \end{bmatrix}_{(x_c, y_c)} \cdot |K| = f(x_c, y_c) \cdot \frac{S_0}{3} \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix},$$

其中 $f(x_c, y_c)$ 是在单元重心处的值 (f 根据具体问题修改).

CODE 4.4. feVector.m

```

1 function fe = feVector(x1,y1,x2,y2,x3,y3)
2 % feVector: element vector
3
4 Am = [1 x1 y1; 1 x2 y2; 1 x3 y3]; A = 1/2*det(Am);
5 xc = 1/3*(x1+x2+x3); yc = 1/3*(y1+y2+y3);
6 % f = @(x,y) (pi^2*y.^2-2).*sin(pi*x);
7 fe = f(xc,yc)*A/3*[1;1;1];
8 end
9 function val = f(x,y)
10    val = (pi^2*y.^2-2).*sin(pi*x);
11 end

```

注 4.3 需要注意的是, 对常函数 $f(x, y) = 1$, 调用匿名函数 $f = @(x, y) 1$ 时, 多个点只会生成一个值, 此时必须改为 $f = @(x, y) 1 * ones(size(x))$ 或 $f = @(x, y) 1 + 0 * x$ (编写函数时也要注意).

4.2.2 sparse 装配法

指标装配很容易用 `sparse` 快速实现, 类似一维问题, 我们先计算出所有单元的刚度矩阵和载荷向量, 且**把它们按行拉直, 逐个单元拼接**, 分别用 K 和 F 表示.

```

1 NT = size(elem,1); % number of nodes and elements
2 Ndof = 3;
3

```

```

4 % All element matrices and vectors
5 K = zeros(NT,Ndof^2); F = zeros(NT,Ndof);
6 for iel = 1:NT
7     % local --> global
8     index = elem(iel,:);
9
10    % element matrix and vector
11    x1 = node(index(1),1); y1 = node(index(1),2);
12    x2 = node(index(2),1); y2 = node(index(2),2);
13    x3 = node(index(3),1); y3 = node(index(3),2);
14
15    ke = feMatrix(x1,y1,x2,y2,x3,y3);
16    fe = feVector(x1,y1,x2,y2,x3,y3);
17
18    K(iel,:) = reshape(ke',Ndof^2,1); % stored in rows
19    F(iel,:) = fe';
20 end

```

所有单元矩阵和载荷向量可用向量法一次生成, 即

```

1 % area of all triangles
2 x1 = node(elem(:,1),1); y1 = node(elem(:,1),2);
3 x2 = node(elem(:,2),1); y2 = node(elem(:,2),2);
4 x3 = node(elem(:,3),1); y3 = node(elem(:,3),2);
5 A = 0.5*((x2.*y3-x3.*y2)-(x1.*y3-x3.*y1)+(x1.*y2-x2.*y1));
6
7 % All element matrices
8 k11 = (x3-x2).^2+(y2-y3).^2;
9 k12 = (x3-x2).*(x1-x3)+(y2-y3).*(y3-y1);
10 k13 = (x3-x2).*(x2-x1)+(y2-y3).*(y1-y2);
11 k21 = k12;
12 k22 = (x1-x3).^2+(y3-y1).^2;
13 k23 = (x1-x3).*(x2-x1)+(y3-y1).*(y1-y2);
14 k31 = k13;
15 k32 = k23;
16 k33 = (x2-x1).^2+(y1-y2).^2;
17 K = [k11,k12,k13,k21,k22,k23,k31,k32,k33];
18 K = K./(4*repmat(A,1,Ndof^2));
19
20 % All vectors

```

```

21 xc = 1/3*(x1+x2+x3); yc = 1/3*(y1+y2+y3);
22 F1 = rhs(xc,yc).*A./3; F2 = F1; F3 = F1;
23 F = [F1,F2,F3];

```

与 CODE 1.2 一样, 我们可以给出 sparse 装配指标, 只要把 Ndof = 2 改为 Ndof=3, 即

CODE 4.5. sparse 装配指标

```

1 % local --> global
2 nnz = NT*Ndof^2;
3 ii = zeros(nnz,1); jj = zeros(nnz,1); ss = K(:, );
4 id = 0;
5 for i = 1:Ndof
6     for j = 1:Ndof
7         ii(id+1:id+NT) = elem(:,i); % zi
8         jj(id+1:id+NT) = elem(:,j); % zj
9         id = id + NT;
10    end
11 end

```

最终的装配程序如下

CODE 4.6. Get_kk_ff_Sparse.m

```

1 function [kk,ff] = Get_kk_ff_Sparse(node,elem)
2
3 N = size(node,1); NT = size(elem,1); Ndof = 3;
4
5 % area of all triangles
6 x1 = node(elem(:,1),1); y1 = node(elem(:,1),2);
7 x2 = node(elem(:,2),1); y2 = node(elem(:,2),2);
8 x3 = node(elem(:,3),1); y3 = node(elem(:,3),2);
9 A = 0.5*((x2.*y3-x3.*y2)-(x1.*y3-x3.*y1)+(x1.*y2-x2.*y1));
10
11 % All element matrices
12 k11 = (x3-x2).^2+(y2-y3).^2;
13 k12 = (x3-x2).*(x1-x3)+(y2-y3).*(y3-y1);
14 k13 = (x3-x2).*(x2-x1)+(y2-y3).*(y1-y2);
15 k21 = k12;
16 k22 = (x1-x3).^2+(y3-y1).^2;
17 k23 = (x1-x3).*(x2-x1)+(y3-y1).*(y1-y2);
18 k31 = k13;

```

```

19 k32 = k23;
20 k33 = (x2-x1).^2+(y1-y2).^2;
21 K = [k11,k12,k13,k21,k22,k23,k31,k32,k33];
22 K = K./(4*repmat(A,1,Ndof^2));
23
24 % All vectors
25 xc = 1/3*(x1+x2+x3); yc = 1/3*(y1+y2+y3);
26 F1 = rhs(xc,yc).*A./3; F2 = F1; F3 = F1;
27 F = [F1,F2,F3];
28
29 % local --> global
30 nnz = NT*Ndof^2;
31 ii = zeros(nnz,1); jj = zeros(nnz,1); ss = K(:);
32 id = 0;
33 for i = 1:Ndof
34     for j = 1:Ndof
35         ii(id+1:id+NT) = elem(:,i); % zi
36         jj(id+1:id+NT) = elem(:,j); % zj
37         id = id + NT;
38     end
39 end
40
41 % stiffness matrix and load vector
42 kk = sparse(ii,jj,ss,N,N);
43 ff = accumarray(elem(:,1), F(:,1), [N 1]);
44 end
45
46 function val = f(x,y)
47     val = (pi^2*y.^2-2).*sin(pi*x);
48 end

```

4.3 边界积分的处理

4.3.1 等效拉平法

设区域 Ω 割分后所得区域为 Ω_h , 变分问题实际上是在 Ω_h 上考虑.

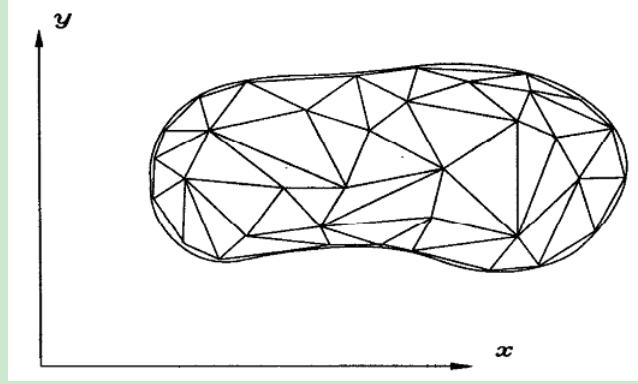


图 4.4. 区域剖分

在不考虑边界条件情况下, 整体变分形式为

$$\int_{\Omega_h} \nabla u \cdot \nabla v dx dy = \int_{\Omega_h} f v dx dy + \int_{\partial \Omega_h} \frac{\partial u}{\partial n} v ds \quad \forall v \in V_h.$$

注意, 前面装配给出的是不考边界积分的部分, 即

$$\int_{\Omega_h} \nabla u \cdot \nabla v dx dy \sim \int_{\Omega_h} f v dx dy \quad \forall v \in V_h,$$

我们还需要加上边界积分

$$\int_{\partial \Omega_h} \frac{\partial u}{\partial n} v ds$$

的贡献. 设边界 $\partial \Omega_h = e_1 \cup e_2 \cup \dots \cup e_l$, 则

$$\int_{\partial \Omega_h} \frac{\partial u}{\partial n} v ds = \sum_i \int_{e_i} \frac{\partial u}{\partial n} v ds, \tag{4.2}$$

这里边界项 $\int_{e_i} \frac{\partial u}{\partial n} v ds$ 相当于一维问题的 $u'v|_{x_{i-1}}^{x_i}$, 而 $\int_{\partial \Omega} \frac{\partial u}{\partial n} v ds$ 相当于 $u'v|_0^1$ (见式 (1.3)).

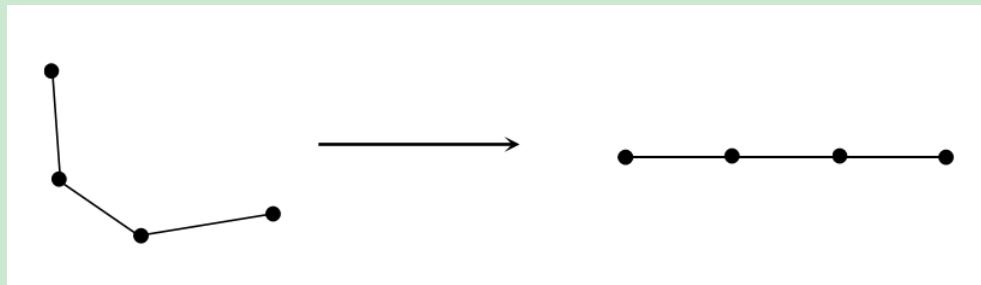


图 4.5. 边界“拉平”

值得注意的是, 若把 $\partial \Omega_h$ “拉平” (想象成一条直线), 则边界上的积分就相当于一维问题的有限元积分, 这是我们处理高维问题的重要观察. 事实上, 设整体变量排列为 u_1, u_2, \dots, u_M , 则有相应的整体节点基为 $\Phi_1, \Phi_2, \dots, \Phi_M$, 满足 $\Phi_j(x_i) = \delta_{ij}$ (x_i 是整体节点). 不妨设前 l 个为全部边界点, 则

$$\Psi_j = \Phi_j|_{\partial \Omega_h}, \quad j = 1, 2, \dots, l$$

相当于“拉平”边界对应的整体节点基(一维问题),因为它满足节点基的定义. 而 Ψ_j 限制在边界单元上就会产生类似一维问题的局部节点基 ϕ_1, ϕ_2 .

这样,对(4.2)就可以类似一维问题那样分单元装配载荷向量.

4.3.2 边界积分的装配

方程组右端的第 j 行要加上

$$v = \Phi_j : \int_{\partial\Omega_h} \frac{\partial u}{\partial n} v ds = \sum_i \int_{e_i} \frac{\partial u}{\partial n} v ds,$$

前面已经分析了它可以类似一维问题处理,下面具体讨论之.

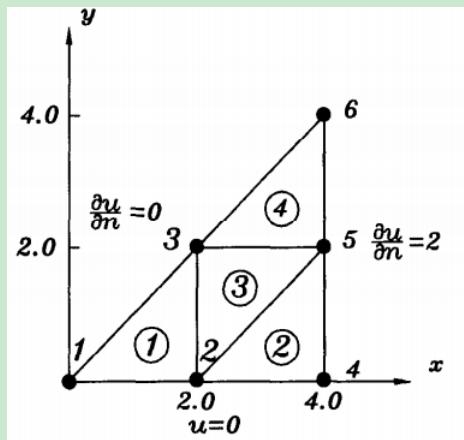


图 4.6. 单元剖分(圈表示单元编号)

为了方便,我们用图 4.6 的区域剖分来说明边界积分的影响,这里边界条件包含 Dirichlet 边界条件和 Neumann 边界条件.

- 先看 Neumann 边界 4-5, 记为 Γ^{45} , 计算 $\int_{\Gamma^{45}} \frac{\partial u}{\partial n} v ds$.

此时通量 $\frac{\partial u}{\partial n} = g_N$ 已知,从而积分可以具体算出,需要把算出的值放到方程的右端(载荷向量中). 单元为 $[x_4, x_5]$, 我们有局部整体对应

$$\{1, 2\} \text{ (local)} \quad \rightarrow \quad \{4, 5\} \text{ (global)},$$

从而单元积分应贡献的行如下

$$\begin{bmatrix} \int_{\Gamma^{45}} g_N \phi_1 ds \\ \int_{\Gamma^{45}} g_N \phi_2 ds \end{bmatrix} \rightarrow \begin{bmatrix} F_4 \\ F_5 \end{bmatrix},$$

其中 ϕ_1, ϕ_2 是局部节点基. 令指标

$$\text{index}(1) = 4; \text{ index}(2) = 5;$$

则可利用载荷向量的装配算法计算. 对 Γ^{56} 类似处理.

- 再看 Dirichlet 边界 1-2, 记为 Γ^{12} .

本来是需要计算积分 $\int_{\Gamma^{12}} \frac{\partial u}{\partial n} v ds$, 此时 $\frac{\partial u}{\partial n}$ 并不知道. 若要计算该积分, 则它产生的是关于 u_1, u_2 的结果, 因而是变量, 需要合并到左侧的刚度矩阵中, 而不是右侧. 但是, u_1, u_2 的值是已知的, 它们对应的行最后都会用恒等式 $u_1 = u(x_1)$ 和 $u_2 = u(x_2)$ 替换, 所以没有必要计算积分.

根据以上分析, 可以给出如下的编程策略.

二维问题的有限元编程策略

- (1) 不考虑边界条件或边界积分项, 按 4.2 节求出暂时的整体刚度矩阵和整体载荷向量.
- (2) 确定 Neumann 边界, 如图 4.6 的 4-5-6 部分, 把这部分边界视为一维问题的整体区域, $\frac{\partial u}{\partial n} = g_N$ 视为一维问题的载荷 f , 从而给出“边界单元载荷向量”, 然后加到整体载荷向量的对应位置上 (若 $g_N = 0$, 则贡献为 0, 直接忽略).
- (3) 确定 Dirichlet 边界, 用“恒等式法”取代对应边界点的系统方程的行.

注 4.4 必须等所有装配过程完成后才能应用 Dirichlet 边界条件, 而 Neumann 边界条件本身是装配问题, 所以必须先处理 Neumann 边界条件, 最后处理 Dirichlet 边界条件.

注 4.5 Neumann 边界对应边界积分, 它可以逐段积分, 而且与积分单元的次序无关, 只要保证每个单元的定向正确即可.

4.3.3 边界单元的积分计算

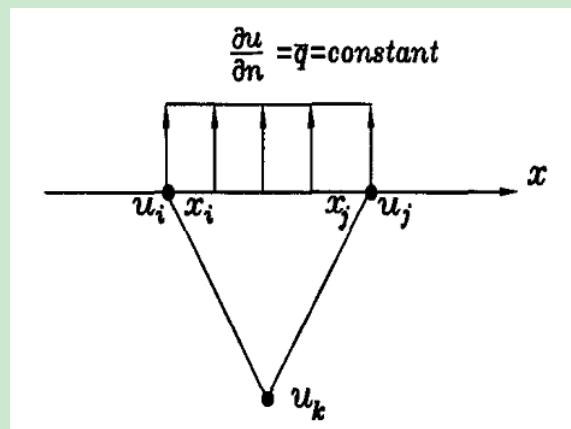


图 4.7. 具常通量的边界单元

- 当 $\frac{\partial u}{\partial n} = \bar{q}$ 为常数时, 如图 4.7 所示. 若三角形的边界单元平行于 x 轴, 则 “边界单元载荷向量” 为

$$\begin{bmatrix} \int_{\Gamma^e} \frac{\partial u}{\partial n} \phi_1 ds \\ \int_{\Gamma^e} \frac{\partial u}{\partial n} \phi_2 ds \end{bmatrix} = \bar{q} \begin{bmatrix} \int_{x_i}^{x_j} \phi_1 dx \\ \int_{x_i}^{x_j} \phi_2 dx \end{bmatrix} = \bar{q} \begin{bmatrix} \int_{x_i}^{x_j} \frac{x_j - x}{x_j - x_i} dx \\ \int_{x_i}^{x_j} \frac{x - x_i}{x_j - x_i} dx \end{bmatrix} = \frac{\bar{q} h_{ij}}{2} \begin{bmatrix} 1 \\ 1 \end{bmatrix},$$

其中 $h_{ij} = x_j - x_i$ 是边界单元的长度.

- 对不是平行于 x 轴的边界, 只要 \bar{q} 为常数, 上面结果就不变 (当然 h_{ij} 要对应理解).
- 对一般情况可用梯形公式近似 (或用中心格式), 例如

$$\int_{\Gamma^e} \frac{\partial u}{\partial n} \phi_1 ds = \frac{h_{ij}}{2} \left(\frac{\partial u}{\partial n}(z_1) \phi_1(z_1) + \frac{\partial u}{\partial n}(z_2) \phi_1(z_2) \right) = \frac{h_{ij}}{2} \frac{\partial u}{\partial n}(z_1) = \frac{h_{ij} q(z_1)}{2},$$

$$\int_{\Gamma^e} \frac{\partial u}{\partial n} \phi_2 ds = \frac{h_{ij}}{2} \left(\frac{\partial u}{\partial n}(z_1) \phi_2(z_1) + \frac{\partial u}{\partial n}(z_2) \phi_2(z_2) \right) = \frac{h_{ij}}{2} \frac{\partial u}{\partial n}(z_2) = \frac{h_{ij} q(z_2)}{2},$$

其中

$$h_{ij} = |z_j - z_i| = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}.$$

注 4.6 h_{ij} 是单元长度而不是代数长度, 这是因为边界积分可以看成拉平后的一维积分, 因是逆时针排列, 故拉平后是从左到右, 因而是单元长度.

4.3.4 边界条件的程序实现

Neumann 边界条件可视为一维问题载荷向量的装配, 为此要给出 Neumann 边界单元“拉平”后从左至右的节点编号 (这里的从左到右意思是按逆时针方向给出的). 前面已经给出, 为 elemN.

Neumann 边界条件可如下快速装配

```

1 % Assemble the Neumann boundary conditions
2 z1 = node(elemN(:,1),:); % coordinates of starting points
3 z2 = node(elemN(:,2),:); % coordinates of ending points
4 h = sqrt(sum((z2-z1).^2,2));
5 F1 = h./2.*g_N(z1); F2 = h./2.*g_N(z2);
6 FN = [F1,F2];
7 ffN = accumarray(elemN(:,1), FN(:,1), [N 1]);

```

其中, g_N 是 Neumann 边界的函数文件, 取为

```

1 function val = g_N(x,y)
2     val = pi*y.^2.*cos(pi*x);
3 end

```

Dirichlet 条件类似一维问题处理

```
1 % ----- Dirichlet boundary conditions -----
2 g_D = @ (x,y) y.^2.*sin(pi*x);
3 N = size(node,1);
4 isBdNode = false(N,1); isBdNode(eD) = true;
5 bdNode = find(isBdNode); freeNode = find(~isBdNode);
6 u = zeros(N,1); u(bdNode) = g_D(node(bdNode,1),node(bdNode,2));
7 ff = ff - kk*u;
```

最后的方程组可如下求解

```
1 u(freeNode) = kk(freeNode,freeNode)\ff(freeNode);
```

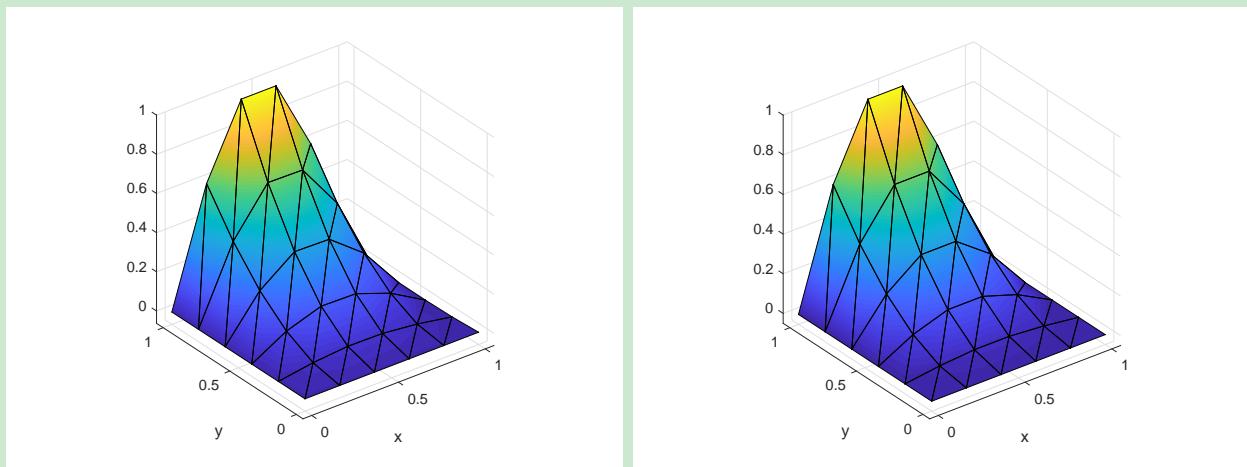
4.3.5 解的图示

MATLAB 早期版本, 例如 MATLAB R2013a 可以用如下语句画图

```
1 p = node';
2 figure, pdesurf(p,t,u);
```

内部主要是使用 pdeplot 画图的. 新版本有所改动, 已不支持上面的方式, 这是因为, MATLAB 网格数据中连通性 t 还有额外的信息. 我们编写一个函数 showsolution.m, 它使用 patch 画图, 下一章详细说明. 用法如下:

```
figure, showsolution(node, elem, u);
```



(a) 数值解

(b) 精确解

图 4.8. 模型问题的数值解与精确解

为了便于对比, 这里对画图的区域做了界定, 特别是 z 轴方向. 如果想去掉这些界定, 例如画绝对误差图, 那么可在画图语句后添加 `zlim('auto')`, 如下图

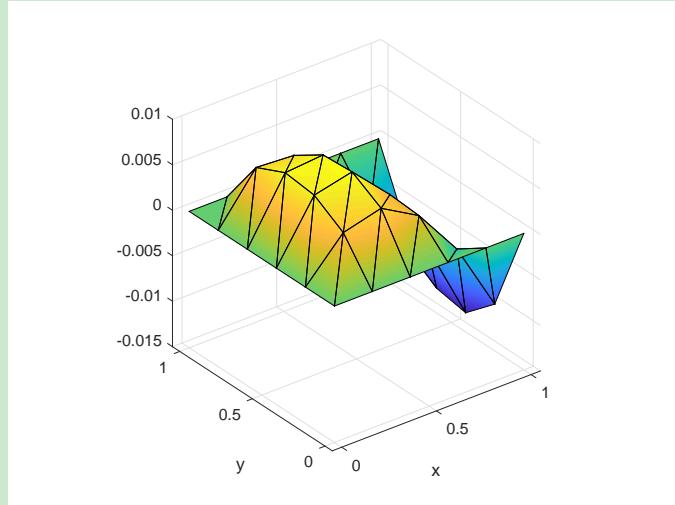


图 4.9. 绝对误差

定义整体相对误差

$$\text{Err} = \frac{\|U - u\|_2}{\|u\|_2},$$

下表给出不同划分的结果.

表 4.1. 整体相对误差 (M 是横向划分, N 是纵向划分)

M	$N = 10$	$N = 20$	$N = 50$
5	1.3931e-02	1.2994e-02	1.2098e-02
10	3.9026e-03	3.3002e-03	2.8684e-03
20	2.2352e-03	9.8932e-04	7.5369e-04

注 4.7 可以计算有限元分析中常用的一些范数, 这里暂时不考虑.

4.4 二维问题程序整理

我们按如下顺序组织程序.

步 1. 建立区域的网格剖分, 确定边界条件, 编写函数为

```
[node, elem, eD, elemN] = meshLap();
```

步 2. 给定 PDE 的相关信息, 包括右端、边界条件及精确解. 我们用结构体存储这些函数的句柄, 编写函数为 (参考 iFEM)

```
pde = pdedata();
```

结构体 pde 的信息如下

```
pde = struct('uexact',@uexact, 'f', @f, 'g_N', @g_N, 'g_D', @g_D);
```

步 3. 计算并装配刚度矩阵和载荷向量, 编写函数为

```
[kk,ff] = Get_kk_ff(node,elem,pde);
```

步 4. 编写主程序 Laplace2D.m, 并在其中处理 Neumann 和 Dirichlet 边界条件.

主程序 Laplace2D.m 如下

CODE 4.7. Laplace2D.m (主程序)

```
1 clc;clear;close all;
2 % ----- Mesh and boundary conditions -----
3 [node,elem,eD,elemN] = meshLap();
4 N = size(node,1);
5
6 % -----
7 % pde = struct('uexact',@uexact, 'f', @f, 'g_N', @g_N, 'g_D', @g_D);
8 pde = pdedata();
9
10 % -----
11 [kk,ff] = Get_kk_ff(node,elem,pde);
12
13 % -----
14 z1 = node(elemN(:,1),:); % starting points
15 z2 = node(elemN(:,2),:); % ending points
16 h = sqrt(sum((z2-z1).^2,2));
17 g_N = pde.g_N;
18 F1 = h./2.*g_N(z1); F2 = h./2.*g_N(z2);
19 FN = [F1,F2];
20 ff = ff + accumarray(elemN(:,1), FN(:,1));
21
22 % -----
23 g_D = pde.g_D;
24 N = size(node,1);
25 isBdNode = false(N,1); isBdNode(eD) = true;
26 bdNode = find(isBdNode); freeNode = find(~isBdNode);
27 pD = node(bdNode,:);
```

```

28 u = zeros(N,1); u(bdNode) = g_D(pD);
29 ff = ff - kk*u;
30
31 % ----- Solve the linear system -----
32 u(freeNode) = kk(freeNode,freeNode)\ff(freeNode);
33
34 % ----- error analysis -----
35 uexact = pde.uexact;
36 ue = uexact(node);
37 format shorte
38 Err = norm(u-ue)/norm(ue)
39
40 figure, showsolution(node, elem, u);
41 figure, showsolution(node, elem, ue);
42
43 Eabs = u-ue; % Absolute errors
44 figure, showsolution(node, elem, Eabs);
45 zlim('auto');

```

步1 网格剖分及边界条件的函数 meshLap.m 如下

CODE 4.8. meshLap.m

```

1 function [node,elem,eD,elemN] = meshLap()
2
3 a1 = 0; b1 = 1; a2 = 0; b2 = 1; Nx = 5; Ny = 5;
4 square = [a1 b1 a2 b2]; h1 = (b1-a1)/Nx; h2 = (b2-a2)/Ny;
5 [node,elem] = squaremesh(square,h1,h2);
6 showmesh(node,elem); findnode(node); findelem(node,elem);
7
8 eps = 1e-4;
9 % ----- Set Dirichlet boundaries -----
10 idL = find(abs(node(:,1)-a1)≤eps); % left
11 idD = find(abs(node(:,2)-a2)≤eps); % down
12 idU = find(abs(node(:,2)-b2)≤eps); % upper
13 id = [idL;idD;idU];
14 eD = unique(id); % node id of Dirichlet boundaries
15
16 % ----- Set Neumann boundaries -----
17 idL = find(abs(node(:,1)-b1)≤eps); % right
18 yn = node(idL,2);

```

```

19 [~,m] = sort(yn); % sort in ascending order
20 id = idL(m);
21 % Neumann elements
22 nel = length(id)-1; % number of Nuemann boundaries
23 elemN = zeros(nel,2);
24 % id of starting and ending points
25 elemN(:,1) = id(1:end-1); elemN(:,2) = id(2:end);

```

步 2 的 PDE 信息函数为

CODE 4.9. pdedata.m

```

1 function pde = pdedata()
2
3 pde = struct('uexact',@uexact, 'f', @f, 'g_N', @g_N, 'g_D', @g_D);
4
5 % exact solution
6 function val = uexact(p)
7     x = p(:,1); y = p(:,2);
8     val = y.^2.*sin(pi*x);
9 end
10 % right side hand function
11 function val = f(p)
12     x = p(:,1); y = p(:,2);
13     val = (pi^2*y.^2-2).*sin(pi*x);
14 end
15 % Neumann boundary conditions
16 function val = g_N(p)
17     x = p(:,1); y = p(:,2);
18     val = pi*y.^2.*cos(pi*x);
19 end
20 % Dirichlet boundary conditions
21 function val = g_D(p)
22     val = uexact(p);
23 end
24 end

```

步 3 的装配程序如下

CODE 4.10. Get_kk_ff.m

```

1 function [kk,ff] = Get_kk_ff(node,elem,pde)

```

```

2
3 N = size(node,1); NT = size(elem,1); Ndof = 3;
4
5 % area of all triangles
6 x1 = node(elem(:,1),1); y1 = node(elem(:,1),2);
7 x2 = node(elem(:,2),1); y2 = node(elem(:,2),2);
8 x3 = node(elem(:,3),1); y3 = node(elem(:,3),2);
9 A = 0.5*((x2.*y3-x3.*y2)-(x1.*y3-x3.*y1)+(x1.*y2-x2.*y1));
10
11 % All element matrices
12 k11 = (x3-x2).^2+(y2-y3).^2;
13 k12 = (x3-x2).*(x1-x3)+(y2-y3).*(y3-y1);
14 k13 = (x3-x2).*(x2-x1)+(y2-y3).*(y1-y2);
15 k21 = k12;
16 k22 = (x1-x3).^2+(y3-y1).^2;
17 k23 = (x1-x3).*(x2-x1)+(y3-y1).*(y1-y2);
18 k31 = k13;
19 k32 = k23;
20 k33 = (x2-x1).^2+(y1-y2).^2;
21 K = [k11,k12,k13,k21,k22,k23,k31,k32,k33];
22 K = K./(4*repmat(A,1,Ndof^2));
23
24 % All vectors
25 f = pde.f;
26 xc = 1/3*(x1+x2+x3); yc = 1/3*(y1+y2+y3); pc = [xc,yc];
27 F1 = f(pc).*A./3; F2 = F1; F3 = F1;
28 F = [F1,F2,F3];
29
30 % local --> global
31 nnz = NT*Ndof^2;
32 ii = zeros(nnz,1); jj = zeros(nnz,1); ss = K(:);
33 id = 0;
34 for i = 1:Ndof
35     for j = 1:Ndof
36         ii(id+1:id+NT) = elem(:,i); % zi
37         jj(id+1:id+NT) = elem(:,j); % zj
38         id = id + NT;
39     end
40 end
41
```

```
42 % stiffness matrix and load vector
43 kk = sparse(ii,jj,ss,N,N);
44 ff = accumarray(elem(:), F(:,[N 1]));
45 end
```

第五章 自适应有限元方法

5.1 自适应方法简介

5.1.1 后验误差估计

考虑 Poisson 方程的 Dirichlet 问题

$$\begin{cases} -\Delta u = f & \text{in } \Omega, \\ u = g & \text{on } \partial\Omega, \end{cases} \quad (5.1)$$

为了方便, 本文只考虑 $g = 0$ 的情形. 变分问题为: 找 $u \in V = H_0^1(\Omega)$ 使得

$$a(u, v) = \ell(v), \quad v \in V,$$

式中,

$$a(u, v) = \int_{\Omega} \nabla u \cdot \nabla v \, dx, \quad \ell(v) = \int_{\Omega} f v \, dx.$$

线性有限元方法为: 找 $u_h \in V_h$ 使得

$$a(u_h, v) = \ell(v), \quad v \in V_h, \quad (5.2)$$

这里, 有限元空间

$$V_h = \left\{ v \in H^1(\Omega) \cap C(\bar{\Omega}) : \quad v|_K \in \mathbb{P}_1(K) \quad \forall K \in \mathcal{T}_h, \quad v|_{\partial\Omega} = 0 \right\},$$

而 \mathcal{T}_h 是满足 shape regularity 条件的三角剖分, 这等价于最小角条件.

对协调有限元问题 (5.2), 利用 Ceá 引理, 误差估计转化为逼近算子 (例如插值算子) 的误差估计, 通常可建立如下的先验估计

$$\|u - u_h\|_{0,K} + h_K |u - u_h|_{1,K} \lesssim h_K^{s+1} |u|_{s+1,K}, \quad s \geq 0.$$

这里, 右端依赖于未知函数 u 的先验信息 (如 H^2 正则性), 因而称为先验估计.

后验误差估计是建立如下类型的估计

$$|||u - u_h||| \lesssim \eta(u_h),$$

式中, $|||\cdot|||$ 是某种范数或半范数, 用以度量误差, η 是与未知函数 u 无关的上界, 且与数值解 u_h 相关, 即 $\eta = \eta(u_h)$. 对问题 (5.2), 我们可建立如下的估计

$$|u - u_h|_1 \lesssim \eta(u_h),$$

式中,

$$\eta = \left(\sum_{K \in \mathcal{T}_h} \eta_K^2 \right)^{1/2}, \quad \eta_K^2 = h_K^2 \|f + \Delta u_h\|_{0,K}^2 + \sum_{e \in \partial K} h_e \|[\partial_{n_e} u_h]\|_{0,e}^2.$$

这里, $f + \Delta u_h$ 显然是数值解关于方程的残差, 注意, 对线性元 $\Delta u_h = 0$, 而

$$[\partial_{n_e} u_h] = \partial_{n_e} u_h|_{K_1} - \partial_{n_e} u_h|_{K_2}$$

为 $\partial_{n_e} u_h$ 跨单元时的跳量, n_e 是对应 K_1 的外法向量.

- 对给定的网格剖分 \mathcal{T}_h , 数值解在每个单元上计算的精度是不同的. 有些单元可能计算的好, 而有些单元可能计算的差, 这与区域以及解的正则性等都有关系. 例如, 对 L 型区域的角点附近, 通常网格剖分要细密一点, 否则这部分计算的就相对差一点. 再比如, 若精确解在某点附近有奇异性, 则这部分网格也要密一点.
- η_K 称为局部误差指示子 (local error indicator), 它的大小可用来度量单元 K 上的计算好坏. 当 η_K 较大时, 我们可以对该单元进行局部加密.

为了方便, 对 $\mathcal{M} \subset \mathcal{T}_h$, 定义

$$\eta(u_h, \mathcal{M}) = \left(\sum_{K \in \mathcal{M}} \eta_K^2 \right)^{1/2}.$$

显然, 有 $\eta(u_h) = \eta(u_h, \mathcal{T}_h)$.

5.1.2 加密或标记准则

现在给出一些局部加密的准则, 常用的有以下三种.

1. The error equidistribution strategy (误差均分策略)

给定 $\theta > 1$, 标记所有的单元 K^* , 满足

$$\eta_{K^*} \geq \theta \frac{\text{TOL}}{\sqrt{NT}},$$

式中, TOL 相当于迭代算法中停止的容许值, NT 是单元的总个数.

2. The maximum marking strategy (Babuska-Rheinboldt strategy)

标记所有的单元 K^* , 使得

$$\eta_{K^*} \geq \theta \max_{K \in \mathcal{T}_h} \eta_K, \quad \theta \in (0, 1).$$

3. The Dörfler bulk strategy

标记所有的单元 $K^* \subset \mathcal{M}_h$, 使得

$$\eta^2(u_h, \mathcal{M}_h) \geq \theta\eta^2, \quad \theta \in (0, 1),$$

这里, \mathcal{M}_h 是 \mathcal{T}_h 的子集. 注意, 通常将 η_K^2 从大到小排列, 找到最少的前若干项单元作为 \mathcal{M}_h .

5.1.3 有限元程序

为了方便操作, 我们先给出一个初始网格上的问题. 考虑 Dirichlet 问题 (5.1), 精确解设为

$$u(x, y) = xy(1 - x)(1 - y)\exp(-1000((x - 0.5)^2 + (y - 0.117)^2)),$$

区域 $\Omega = (0, 1)^2$, 此时 $g = 0$. 这个解衰减得很快, 它在 $(0.5, 0.117)$ 处的值要明显大于其他处, 为此真正计算过程中, 只需要该点附近的网格细密就可 (不妨称为奇点).

根据二维问题的说明, 我们很容易给出有限元程序. 所有计算的过程编写成函数文件 Poisson.m, 如下使用 (见 GitHub 上传文件)

```
u = Poisson(node, elem, bdFlag, pde);
```

这里, `bdFlag` 是结构体, 用以存储边界条件信息 (见下面的 `setboundary.m`). `pde` 是方程信息, 由 `pdedata.m` 函数给出. 为了方便, 我们定义一个设置边界的函数 `setboundary.m`, 如下

```

1 function bdFlag = setboundary(node, a1, b1, a2, b2)
2
3 eps = 1e-4;
4 % ----- Set Dirichlet boundaries -----
5 idL = find(abs(node(:,1)-a1)≤eps); % left
6 idR = find(abs(node(:,1)-b1)≤eps); % right
7 idD = find(abs(node(:,2)-a2)≤eps); % down
8 idU = find(abs(node(:,2)-b2)≤eps); % upper
9 id = [idL; idR; idD; idU];
10 eD = unique(id); % node id of Dirichlet boundaries
11
12 % ----- Set Neumann boundaries -----
13 elemN = [];
14
15 bdFlag.eD = eD; bdFlag.elemN = elemN;

```

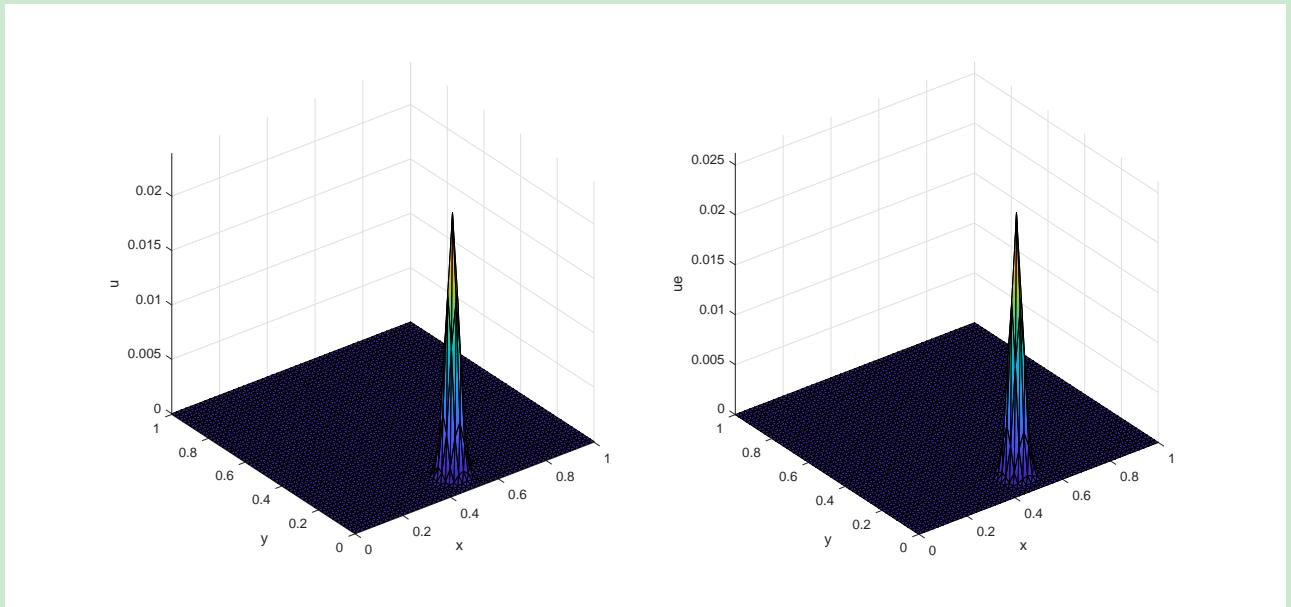
如下是主程序 (`mainFEM.m`)

```

1 clc;clear;close all;
2 % ----- Mesh and PDE -----
3 a1 = 0; b1 = 1; a2 = 0; b2 = 1;
4 Nx = 50; Ny = 50; h1 = 1/Nx; h2 = 1/Ny;
5 [node,elem] = squaremesh([a1 b1 a2 b2],h1,h2);
6 pde = pdedata();
7 bdFlag = setboundary(node,a1,b1,a2,b2);
8
9 % ----- solve Poisson problem -----
10 u = Poisson(node,elem,bdFlag,pde);
11
12 % ----- error analysis -----
13 ueexact = pde.ueexact;
14 ue = ueexact(node);
15 figure,
16 subplot(1,2,1), showsolution(node,elem,u);
17 zlabel('u');
18 subplot(1,2,2), showsolution(node,elem,ue);
19 zlabel('ue');
20 Eabs = u-ue; % Absolute errors
21 figure, showsolution(node,elem,Eabs); zlim('auto');

```

横纵划分 50 等份的结果如下



5.2 误差指示子的计算

现在考虑误差指示子

$$\eta_K^2 = h_K^2 \|f\|_{0,K}^2 + \sum_{e \in \partial K} h_e \|[\partial_{n_e} u_h]\|_{0,e}^2$$

的计算, 这里右端第一项对应残差, 记为 elemRes; 第二项对应跳量, 记为 elemJump. 注意它们都是 NT 个分量的向量, 每个位置对应一个单元.

5.2.1 残差的计算

右端第一部分是三角形单元上的积分, 有相应的 Gauss 求积公式. iFEM 中提供了三角形上的 Gauss 求积节点与权重, 即 quadpts.m. 我们简单说明一下其用法. 实际上, 那里的权重和节点是针对面积坐标下的参考三角形 \hat{T} 进行的, 积分公式为

$$\iint_{\hat{T}} f(\lambda_1, \lambda_2, \lambda_3) d\hat{\sigma} \approx |\hat{T}| \sum_{i=1}^{n_g} w_i f(\lambda_{1,i}, \lambda_{2,i}, \lambda_{3,i}), \quad |\hat{T}| = \frac{1}{2},$$

其中,

$$\begin{cases} x = x_1 \lambda_1 + x_2 \lambda_2 + x_3 \lambda_3 \\ y = y_1 \lambda_1 + y_2 \lambda_2 + y_3 \lambda_3 \end{cases}, \quad \lambda_1 + \lambda_2 + \lambda_3 = 1, \quad (5.3)$$

注意到

$$\det \left(\frac{\partial(x, y)}{\partial(\lambda_1, \lambda_2)} \right) = 2S,$$

这里 S 是三角形 T 的代数面积, 我们有

$$\int_T F(x, y) d\sigma = 2 |T| \int_{\hat{T}} f(\lambda_1, \lambda_2, \lambda_3) d\hat{\sigma} = |T| \sum_{i=1}^{n_g} w_i f(\lambda_{1,i}, \lambda_{2,i}, \lambda_{3,i}).$$

因变换前后点处的值不变, 故

$$\int_T F(x, y) d\sigma = |T| \sum_{i=1}^{n_g} w_i F(x_i, y_i).$$

利用 (5.3) 可把参考元上的 Gauss 点 $(\lambda_{1,i}, \lambda_{2,i}, \lambda_{3,i})$ 转化为 T 上的点. 可如下计算 $\|f\|_{0,K}^2$:

```

1 % n: n-th order quadrature rule
2 f = pde.f; n = 3;
3 [lambda, weight] = quadpts(n);
4 NT = size(elem,1); fL2 = zeros(NT,1); % ||f||_{0,K}^2
5 for iel = 1:NT
6     vK = node(elem(iel,:)); % vertices of K
7     area = 0.5*abs(det([[1;1;1],vK]));

```

```

8      xy = lambda*vK;
9      fL2(iel) = area*dot(weight,f(xy).^2);
10 end

```

注 5.1 iFEM 中定义 $h_K = |K|^{1/2}$, 这由 shape regularity 保证. 这样, $h_K^2 = |K|$, 再考虑到积分公式前面的 $|K|$, iFEM 中会出现 $h_K^2 \cdot |K| = |K|^2$, 这就是那里的函数 estimateresidual.m 计算时给出如下语句的原因

```
elemResidual = elemResidual.* (area.^2);
```

本文直接计算直径, 单元面积以及直径在前面的 auxstructure.m 中已经给出. 这样, 可如下计算 elemRes.

```

1 % ----- elemRes -----
2 aux = auxstructure(node, elem);
3 area = aux.area; diameter = aux.diameter;
4 f = pde.f;
5 n = 3; [lambda, weight] = quadpts(n);
6 NT = size(elem, 1); elemRes = zeros(NT, 1);
7 for iel = 1:NT
8     vK = node(elem(iel, :), :); % vertices of K
9     xy = lambda*vK;
10    elemRes(iel) = weight*f(xy).^2;
11 end
12 elemRes = diameter.^2.*area.*elemRes;

```

5.2.2 边界跳量的计算

现在计算第二项

$$\sum_{e \in \partial K} h_e \|[\partial_{n_e} u_h]\|_{0,e}^2,$$

其中 $\partial_{n_e} u_h = \nabla u_h \cdot n_e$.

- 给定左侧单元 K_1 及其边 e , 设右侧单元为 K_2 . 再设 K_1 的边界外法向量为 n_e , 则跳量为

$$[\partial_{n_e} u_h] = [\nabla u_h \cdot n_e] = \nabla u_h \cdot n_e|_{K_1} - \nabla u_h \cdot n_e|_{K_2} = (\nabla u_h|_{K_1} - \nabla u_h|_{K_2}) \cdot n_e.$$

- 注意, 对左右单元跳量是一致的, 从而 $h_e \|[\partial_{n_e} u_h]\|_{0,e}^2$ 也是一致的. 注意到 ∇u_h 是常

数, 我们有

$$\begin{aligned} h_e \|[\partial_{n_e} u_h]\|_{0,e}^2 &= h_e \int_e [\nabla u_h \cdot n_e]^2 ds = h_e \int_e [\nabla u_h \cdot n_e]^2 ds \\ &= h_e^2 [\nabla u_h \cdot n_e]^2 = [\nabla u_h \cdot h_e n_e]^2 \end{aligned}$$

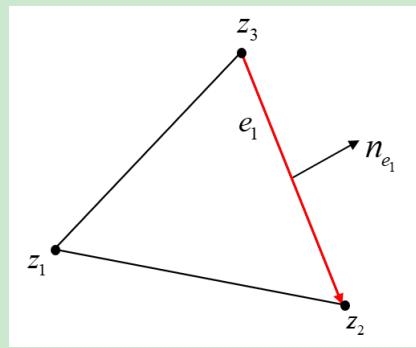
因结果是正的, 我们不必关心 n_e 的方向, 只要取绝对值即可.

- 辅助数据结构中:

- edge 给出了一维边的端点标记, 行索引对应自然序号;
- elem2edge 给出了按单元存储的三条边的自然序号;
- edge2elem, 它给出了边 e 的左右单元.

- 为了避免重复计算, 我们将把所有的一维边 e 的结果计算出来, 然后再给出每个单元的结果.

先考虑 $h_e n_e$ 的计算, 我们按照一维边集合 edge 计算, 结果保存为 nedge.



如图, 三角形 z_i 的对边为 e_i , 相应的外法向量为 n_{e_i} . 可以看到, $h_{e_1} n_{e_1}$ 可由边向量 $e_1 = \overrightarrow{z_3 z_2}$ 逆时针旋转 90° 获得. 易知, 向量 $\alpha = (a, b)$ 旋转 90° 后为 $(-b, a)$. 如下获得所有一维边的 $h_e n_e$ (忽略方向):

```

1 % edge vectors
2 ve = node(edge(:,2),:) - node(edge(:,1),:);
3 % scaled norm vectors he*ne
4 nedge = [-ve(:,2),ve(:,1)]; % stored in rows

```

把 h_e 和 n_e 放在一起, 避免了求边长.

再考虑 ∇u_h 的计算. 注意, 我们已经在初始网格 \mathcal{T}_h 上获得了 u_h (即每个节点处的值, 为了方便, 第 i 个节点的值记为 u_i), 据此可计算 ∇u_h . 设三角形 K 的三个顶点为 z_1, z_2, z_3 , 相应的数值解为 u_1, u_2, u_3 , 则在该单元上有

$$u_h = u_1 \lambda_1 + u_2 \lambda_2 + u_3 \lambda_3,$$

其中, $\lambda_1, \lambda_2, \lambda_3$ 是相应的节点基. 于是,

$$\nabla u_h = u_1 \nabla \lambda_1 + u_2 \nabla \lambda_2 + u_3 \nabla \lambda_3,$$

为此需要计算 $\nabla \lambda_i$. 可以证明,

$$\begin{aligned}\frac{\partial \lambda_1}{\partial x} &= \frac{1}{2|K|}(y_2 - y_3), & \frac{\partial \lambda_1}{\partial y} &= \frac{1}{2|K|}(x_3 - x_2), \\ \frac{\partial \lambda_2}{\partial x} &= \frac{1}{2|K|}(y_3 - y_1), & \frac{\partial \lambda_2}{\partial y} &= \frac{1}{2|K|}(x_1 - x_3), \\ \lambda_3 &= 1 - \lambda_1 - \lambda_2.\end{aligned}$$

左右单元的如下计算

```

1 % information of left and right elements
2 k1 = edge2elem(:,1); k2 = edge2elem(:,2);
3 index1 = elem(k1,:); index2 = elem(k2,:);
4 zL1 = node(index1(:,1),:); zR1 = node(index2(:,1),:);
5 zL2 = node(index1(:,2),:); zR2 = node(index2(:,2),:);
6 zL3 = node(index1(:,3),:); zR3 = node(index2(:,3),:);

7
8 % grad of nodal basis functions
9 gradL1 = [zL2(:,2)-zL3(:,2), zL3(:,1)-zL2(:,1)]; % stored in rows
10 gradL2 = [zL3(:,2)-zL1(:,2), zL1(:,1)-zL3(:,1)];
11 gradR1 = [zR2(:,2)-zR3(:,2), zR3(:,1)-zR2(:,1)];
12 gradR2 = [zR3(:,2)-zR1(:,2), zR1(:,1)-zR3(:,1)];

13
14 gradL1 = gradL1./(2*area(k1)); gradR1 = gradR1./(2*area(k2));
15 gradL2 = gradL2./(2*area(k1)); gradR2 = gradR2./(2*area(k2));
16 gradL3 = -(gradL1+gradL2); gradR3 = -(gradR1+gradR2);

```

这里, 梯度向量按行排列坐标, 每行对应一个边.

注 5.2 设 $A = (a_{ij})$ 为 $m \times n$ 的矩阵, $b = (b_i)$ 是 m 维的列向量, 则

- $A.*b$ 与 $b.*A$ 相同, 都为 $(b_i a_{ij})_{m \times n}$, 即行之间相乘.
- $A./b$ 为 $(a_{ij}/b_i)_{m \times n}$, $b./A$ 为 $(b_i/a_{ij})_{m \times n}$, 都是行之间相除.

据此, 可如下计算每条边 e 对应的 $[\nabla u_h]$

```

1 % grad of uh
2 gradLu = u(index1(:,1)).*gradL1 + u(index1(:,2)).*gradL2 + ...
           u(index1(:,3)).*gradL3;

```

```

3 gradRu = u(index2(:,1)).*gradR1 + u(index2(:,2)).*gradR2 + ...
    u(index2(:,3)).*gradR3;
4 % jump of gradu
5 Jumper = gradLu - gradRu;
6 Jumper(k1==k2,:) = gradLu(k1==k2,:);

```

这里, $k_1 == k_2$ 表示边界上的, 值为本身 (对齐次 Dirichlet 边界部分, 因检验函数在边界上为零, 跳量部分可以去掉, 例如 iFEM. 本文保留).

由此可得到边集合对应的 $h_e \|\partial_{n_e} u_h\|_{0,e}^2$ 的结果, 记为 edgeJump

```

1 % edgeJump
2 edgeJump = dot(Jumper', nedge').^2; edgeJump = edgeJump';

```

注意, 对两个相同的矩阵, `dot` 的作用时对应列进行运算. 因为我们要把每行对应的向量点乘, 故进行转置再 `dot`.

知道每条边的跳量后, 我们可以利用辅助数据结构 `elem2edge` 获得单元的跳量 $\sum_{e \in \partial K} h_e \|\partial_{n_e} u_h\|_{0,e}^2$.

```

1 % elemJump
2 elemJump = sum(edgeJump(elem2edge), 2);

```

`edgeJump` 是一个列向量, 行索引对应边的自然序号. `elem2edge` 存储单元边的自然序号, 它是 $NT \times 3$ 的矩阵, 第 1 列对应每个单元的第 1 条边, 依此类推. `edgeJump(elem2edge)` 是与 `elem2edge` 相同维数的矩阵, 每个位置给出对应边的跳量. 显然按行求和就得到每个单元的跳量.

综上, 误差指示子为

```

1 % ----- Local error indicator -----
2 eta = (elemRes + elemJump).^(1/2);

```

5.2.3 indicator 函数

误差指示子计算的完整程序如下

```

1 function eta = indicator(node, elem, u, pde)
2
3 % ----- auxstructure -----
4 aux = auxstructure(node, elem);
5 elem2edge = aux.elem2edge;
6 edge = aux.edge;

```

```

7 edge2elem = aux.edge2elem;
8 area = aux.area; diameter = aux.diameter;
9
10 % ----- elemRes -----
11 f = pde.f;
12 n = 3; [lambda,weight] = quadpts(n);
13 NT = size(elem,1); elemRes = zeros(NT,1);
14 for iel = 1:NT
15     vK = node(elem(iel,:)); % vertices of K
16     xy = lambda*vK;
17     elemRes(iel) = dot(weight,f(xy).^2);
18 end
19 elemRes = diameter.^2.*area.*elemRes;
20
21
22 % ----- elemJump -----
23 % edge vectors
24 ve = node(edge(:,2))-node(edge(:,1));
25 % scaled norm vectors he*ne
26 nedge = [-ve(:,2),ve(:,1)]; % stored in rows
27
28 % information of left and right elements
29 k1 = edge2elem(:,1); k2 = edge2elem(:,2);
30 index1 = elem(k1,:); index2 = elem(k2,:);
31 zL1 = node(index1(:,1)); zR1 = node(index2(:,1));
32 zL2 = node(index1(:,2)); zR2 = node(index2(:,2));
33 zL3 = node(index1(:,3)); zR3 = node(index2(:,3));
34
35 % grad of nodal basis functions
36 gradL1 = [zL2(:,2)-zL3(:,2), zL3(:,1)-zL2(:,1)]; % stored in rows
37 gradL2 = [zL3(:,2)-zL1(:,2), zL1(:,1)-zL3(:,1)];
38 gradR1 = [zR2(:,2)-zR3(:,2), zR3(:,1)-zR2(:,1)];
39 gradR2 = [zR3(:,2)-zR1(:,2), zR1(:,1)-zR3(:,1)];
40
41 gradL1 = gradL1./(2*area(k1)); gradR1 = gradR1./(2*area(k2));
42 gradL2 = gradL2./(2*area(k1)); gradR2 = gradR2./(2*area(k2));
43 gradL3 = -(gradL1+gradL2); gradR3 = -(gradR1+gradR2);
44
45 % grad of uh
46 gradLu = u(index1(:,1)).*gradL1 + u(index1(:,2)).*gradL2 + ...

```

```

    u(index1(:,3)).*gradL3;
47 gradRu = u(index2(:,1)).*gradR1 + u(index2(:,2)).*gradR2 + ...
    u(index2(:,3)).*gradR3;

48
49 % jump of gradu
50 Jumper = gradLu - gradRu;
51 Jumper(k1==k2,:) = gradLu(k1==k2,:);

52
53 % edgeJump
54 edgeJump = dot(Jumper', nedge').^2; edgeJump = edgeJump';
55
56 % elemJump
57 elemJump = sum(edgeJump(elem2edge), 2);
58
59 % ----- Local error indicator -----
60 eta = (elemRes + elemJump).^(1/2);

```

5.3 标记算法的实现

以下只考虑后两种标记算法, 且假设 η_K 都已获得.

1. The maximum marking strategy

标记所有的单元 K^* , 使得

$$\eta_{K^*} \geq \theta \max_{K \in \mathcal{T}_h} \eta_K, \quad \theta \in (0, 1).$$

程序如下

```

1 NT = size(elem,1); isMark = false(NT,1);
2 isMark(eta > theta * max(eta)) = 1;
3 elemMarked = find(isMark == true);

```

2. The Dörfler bulk strategy

标记所有的单元 $K^* \subset \mathcal{M}_h$, 使得

$$\eta^2(u_h, \mathcal{M}_h) \geq \theta \eta^2, \quad \theta \in (0, 1).$$

将单元按 η_K^2 从大到小排列, 并命名为 K_1, K_2, \dots . 定义

$$s(n) = \sum_{i=1}^n \eta_{K_i}^2,$$

则要找到最小的 n^* 使得,

$$s(n^*) \geq \theta\eta^2 = \theta s(\text{NT}).$$

显然 n_* 存在, 且至多为 NT . 我们可转而找最大的 n_* 使得

$$s(n_*) < \theta\eta^2 = \theta s(\text{NT}). \quad (5.4)$$

这样, K_1, \dots, K_{n_*} 都是需要标记的单元. 自然 $n_* < \text{NT}$, 故 $n^* = n_* + 1$. 为了方便, 我们只标记 K_1, \dots, K_{n_*} . 注意, 若没有满足条件 (5.4) 的 n_* , 则表明 K_1 就是需要的. 程序如下

```

1 [sortedEta,id] = sort(eta.^2,'descend');
2 x = cumsum(sortedEta);
3 isMark(id(x < theta*x(end))) = 1;
4 isMark(id(1)) = 1;
5 elemMarked = find(isMark==true);

```

这里, `cumsum` 是累加函数.

综上, 标记程序如下

CODE 5.1. mark.m

```

1 function elemMarked = mark(elem,eta,theta,method)
2 %Mark mark element.
3
4 NT = size(elem,1); isMark = false(NT,1);
5 if ~exist('method','var'), method = 'Dorfler'; end
6 % default marking is Dofler bulk strategy
7 switch strcmpi(method,'max')
8     case true
9         isMark(eta>theta*max(eta))=1;
10    case false
11        [sortedEta,id] = sort(eta.^2,'descend');
12        x = cumsum(sortedEta);
13        isMark(id(x < theta*x(end))) = 1;
14        isMark(id(1)) = 1;
15    end
16 elemMarked = find(isMark==true);

```

这里, `strcmpi(method, 'max')` 是进行字符串比较且不区分大小写, 因此只要 `method` 是 `max` (不区分大小写), 则选择第一种方法, 其他任何情况都是 Dörfler.

注 5.3 对 maximum marking strategy, θ 越大, 标记的单元越少; Dörfler bulk strategy 恰恰相反.

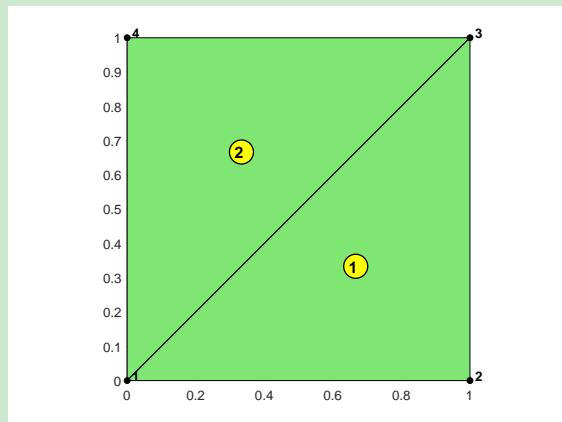
5.4 Newest-node bisection

现在介绍一种局部加密算法, 称为 Newest-node bisection, 不妨翻译为最新点二分. 加密时要注意两点:

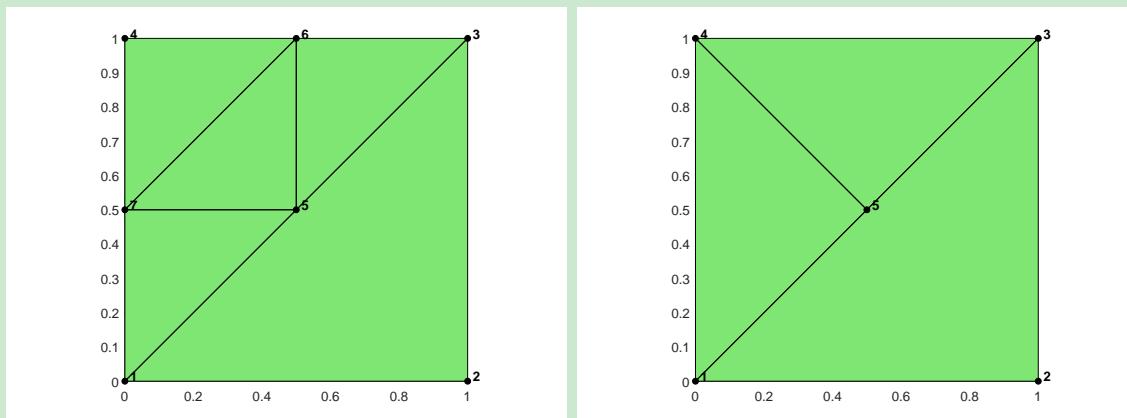
- 保持 shape regularity: 等价于最小角条件;
- 保持协调: 无悬挂点.

5.4.1 局部加密方式

标记好单元后, 我们就要对它们进行局部加密. 以如下的初始网格为例进行说明, 且考虑对单元 2 进行加密.



通常有两种局部加密方式, 如下图所示



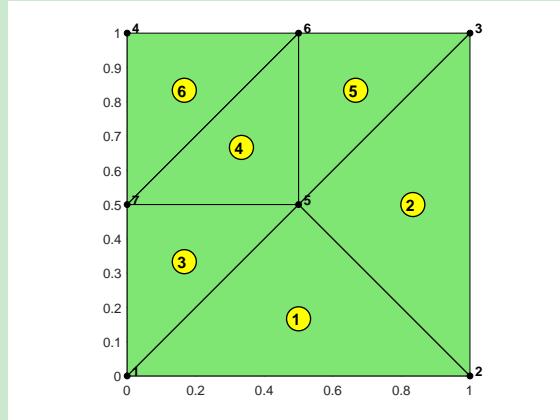
(a) regular refinement

(b) bisection refinement

图 5.1. 两种加密方式

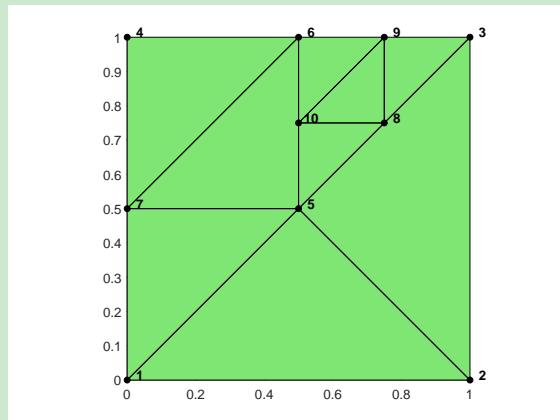
- 正规加密 (regular refinement): 连接三角形各边的中点.
- 二分加密 (bisection refinement): 连接三角形某个顶点与其对边的中点. 称该顶点为三角形的 peak, 对边为 base.

可以看到, 这两种加密均导致非协调剖分, 即含有悬挂点 5. 为了解决悬挂点 5, 对正规加密的图 5.1 (a), 可连接 2-5, 即对单元 ① 进行二分, 如下图

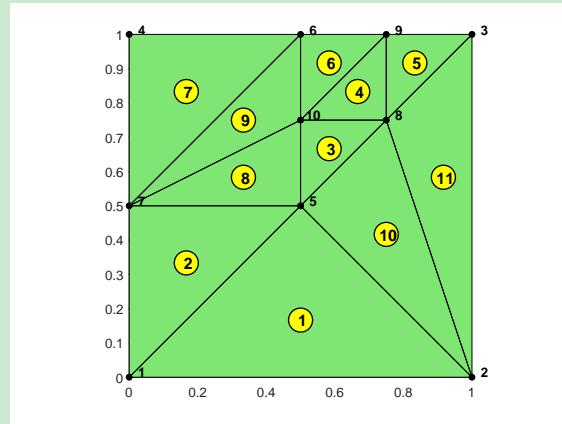


通常称这种处理悬挂点的方法为: the bisection of a triangle as a green refinement. 需要注意的是, 处理正规加密悬挂点的 green refinement 可能导致三角形的退化. 这里退化的意思是剖分不再满足 shape regularity 条件, 因为此时三角形的角可能变得很小.

例如继续对单元 ⑤ 进行正规加密, 得



此时会出现悬挂点 8 和 10, 从而要连接 7-10 和 2-8, 得 green refinement



继续对图中的单元⑤进行加密的话, 单元⑪又被二分, 从而三角形的角会越来越小.

注 5.4 正规加密可以说是一种自然加密, 因为它在加密过程中产生的子三角形与大三角形是相似的, 从而自然地继承了 shape regularity. 本文考虑二分加密.

5.4.2 最新点二分的简单说明

为了简明, 我们用一个例子来说明最新点二分.

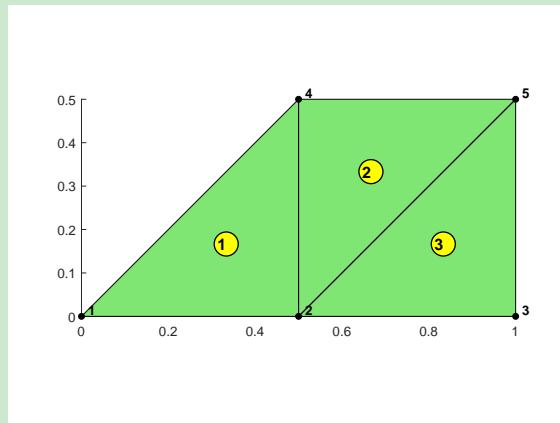


图 5.2. 初始剖分

peak 与 base

如图, 设单元①和③是标记单元, 即 `elemMarked = [1; 3]`. 对一个三角形, 二分是连接某个顶点与对边的中点, 称该顶点为 peak, 对边为 base.

peak 可以是任意一个, 注意到三角形是按顶点逆时针顺序记录的, 循环置换仍表示同一个三角形, 即 `elem(t, [1 2 3])`, `elem(t, [2 3 1])` 和 `elem(t, [3 1 2])` 都表示单元三角形 t . 为此, 我们规定: 第 1 个顶点为 peak, 即 `elem(t, 1)` 是 peak. 根据数据结构的规定, 第 1 个顶点的对边为第 1 条边, 它就是相应的 base, 即为 `elem(t, [2 3])`. 现在假设, 单元①和③的 peak 分别为 1, 3.

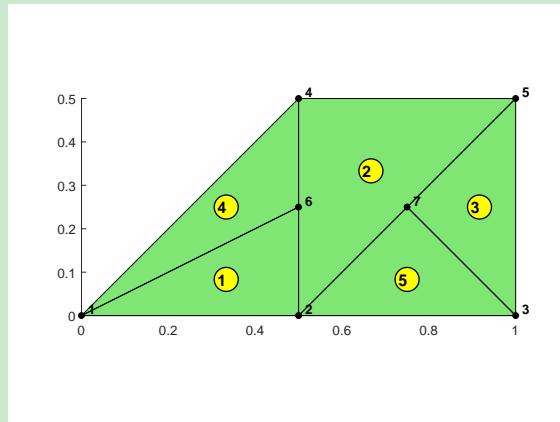
辅助数据结构中, `elem2edge` 按单元存储边的自然序号, 从而 `base` 可用 `elem2edge(t, 1)` 获得自然序号, 这样, 标记单元的 `base` 在边集合中的自然序号为

```
base = elem2edge(elemMarked, 1);
```

在一个三角形中 `peak` 与 `base` 是一一对应的, 只要记录 `base` 即可.

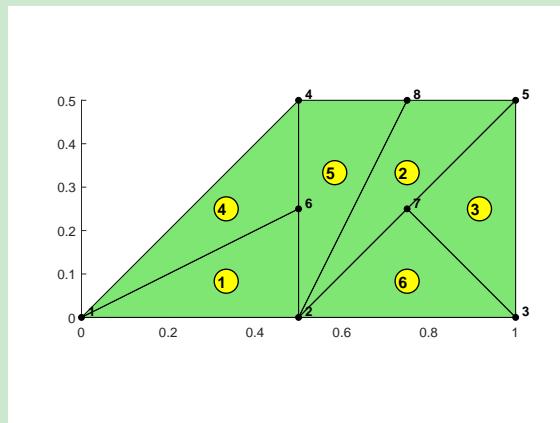
标记单元的扩充

现在开始对标记单元进行二分, 结果为



对每个单元, 规定: 二分的左侧子三角形保留原编号, 右侧子三角形从 $NT+1$ 开始按顺序编号. 为了处理悬挂点, 必须对 `base` 的相邻单元 ② 进行剖分. 这表明, `base` 的左右单元都要标记. 而单元与 `base` 是对应的, 为此只需要记录 `base` 即可.

最新点二分是把 `base` 的中点作为子三角形的 `peak`. 一个重要观察是, 单元 ② 的任一顶点作为 `peak` 都可解决悬挂点问题. 若 4 是 `peak`, 则连接 4-7 后悬挂点 7 消失. 根据规定, 7 是子三角形 7-4-2 的 `peak`. 这样, 单元 1-2-4 与子三角形 7-4-2 有公共的 `base`, 它们分别连接 `peak` 和 `base`, 悬挂点 6 也消失. 若 5 是 `peak`, 则与上面的情形类似. 若 2 是悬挂点, 连接对边后有



此时又划归到有公共 base 的情形, 即子三角形 8-4-2 与单元 1-2-4 有公共 base 2-4, 而子三角形 8-2-5 与单元 3-5-2 有公共 base 5-2.

注 5.5 当解决悬挂点后, 我们不再进行二分. 这保证了 base 都是原来的 edge, 即没有新的边成为 base. 称其为 “base-edge” 性质.

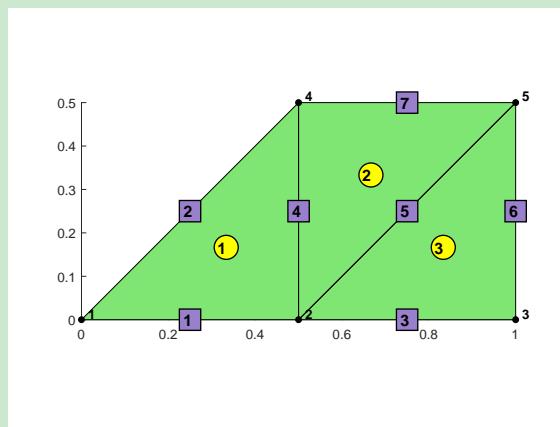
注 5.6 “base-edge” 性质使得我们可方便地进行二次加密, 以去除悬挂点. 例如, 在上图中 base 只有 2-4, 5-4 和 5-2. 对单元 ①, 按规定, 尽管 6 是 peak, 但其对边不在 base 集合中, 从而不会发生二分. 单元 ⑤ 则不同, 它的 peak 为 8, 对边 4-2 在 base 集合中, 因而会二分. 所以, 在对扩展后的标记三角形进行二分后, 再确认每个子三角形 peak 的对边在不在 base 集合中, 就可给出最终的加密网格.

注 5.7 为了方便, 以下称二分标记单元 (base 的左右单元) 为标记二分, 而为了去除悬挂点的二分为协调二分.

5.4.3 标记二分的程序实现

辅助数据 `elem2edge` 按单元存储边的自然序号, 单元的 base 显然为 `elem2edge(:, 1)`. 为了方便使用 `elem2edge`, 下面编程中尽量把与 `edge` 对应.

base 的生成



- base 最多 NE 个, 我们用数组 `isCutEdge` 记录. 给定初始标记单元 `elemMarked ... = [1; 3]`, 则初始的 base 如下获得

```

1  isCutEdge = false(NE, 1);
2  base = elem2edge(elemMarked, 1);
3  isCutEdge(base) = true;

```

其中, `elem2edge(elemMarked, 1)` 给定了标记单元第 1 条边的自然序号, 即标记单元的 base. 例子的 `base = [4; 5]`.

- 接着我们要补充相邻三角形的 base. 为此先找到相邻三角形. 辅助数据结构中给出了 `neighbor` (稀疏矩阵), 其 (i, j) 位置的值表示单元 K_i 的边 e_j 的相邻单元编号, 据此可找到 base 的相邻三角形.

```
refineNeighbor = full(diag(neighbor(elemMarked, base)));
```

注意 `neighbor(elemMarked, base)` 实际上给出的是交错位置矩阵, 取对角线就是需要的. 也可用拉直索引取元素

```
1 cc = NT*(base-1)+elemMarked;
2 refineNeighbor = full(neighor(cc));
```

结果为 `refineNeighbor=[2;2]`. 可以去除重复的.

- 相邻单元的 base 为

```
baseNeighbor = elem2edge(refineNeighbor, 1);
```

例子的 `baseNeighbor=7`. 当然, 其中可能含有已存在的 base, 如下获得新的 base.

```
1 baseNeighbor = elem2edge(refineNeighbor, 1);
2 elemNeighborMarked = refineNeighbor(~isCutEdge(baseNeighbor));
3 elemMarked = elemNeighborMarked;
4 base = elem2edge(elemMarked, 1); isCutEdge(base) = true;
```

这里, `isCutEdge(baseNeighbor)` 确定相邻单元的 base 是否是旧的 base, 对应真的位置. 这样, `~isCutEdge(baseNeighbor)` 真的位置对应新产生的 base. 后面就是获得新产生的 base 所在的单元, 然后重复之前的标记就可获得新的 base.

- 对新单元重复之前的过程即可消除新单元可能存在的悬挂点 (可以证明, 这个过程在有限步内停止).

综上, base 如下生成

```
1 isCutEdge = false(NE, 1);
2 while sum(elemMarked)>0
3     base = elem2edge(elemMarked, 1); isCutEdge(base) = true;
4     cc = NT*(base-1)+elemMarked;
5     refineNeighbor = unique(full(neighor(cc)));
6     baseNeighbor = elem2edge(refineNeighbor, 1);
```

```

7     elemMarked = refineNeighbor(~isCutEdge(baseNeighbor));
8 end

```

`isCutEdge` 就记录了最终的 base. 例子的 `isCutEdge([4, 5, 7])=true`.

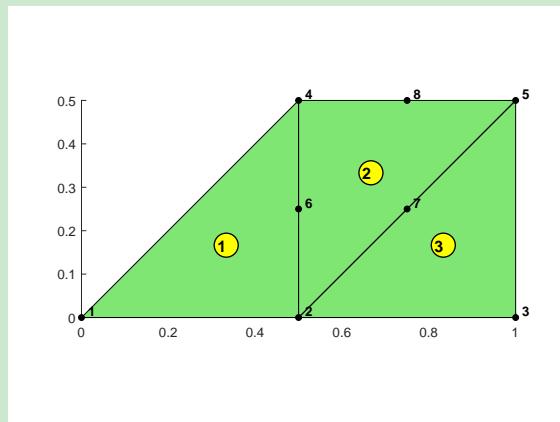
注 5.8 要想实现标记二分, 我们需要确定 base 对应的单元. 回忆一下, `elem2edge(t, 1)` 给出的是单元 t 的 base 的自然序号. 于是, 可用如下语句找到单元

```
t = find(isCutEdge(elem2edge(:, 1))==true);
```

标记二分的基本数据结构

现在获得标记二分的基本数据结构 `node` 和 `elem`.

对 `node`, 只要把标记为 base 的边的中点加入 `node` 即可.



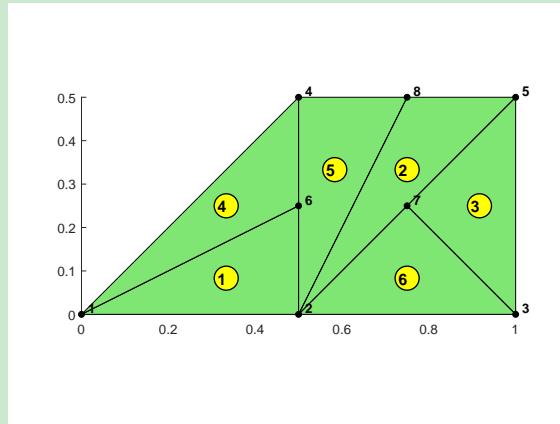
原先顶点个数为 N , base 的个数为 $N_{base}=\text{sum}(\text{isCutEdge})$. 于是可如下增加节点

```

1 % ----- Add new nodes -----
2 Nbase = sum(isCutEdge);
3 edgeCutNumber = N + (1:Nbase);
4 edgebase = edge(isCutEdge,:);
5 node(edgeCutNumber,:) = (node(edgebase(:,1),:)+...
    node(edgebase(:,2),:))/2;

```

接着, 我们要连接 peak 和 base, 获得如下单元分布



```

1 % ----- Refine elements -----
2 edgeCutNumber = zeros(NE,1);
3 edgeCutNumber(isCutEdge) = N + (1:Nbase);
4 t = find(isCutEdge(elem2edge(:,1))==true);
5 newNT = length(t);
6 if newNT>0
7     L = t; R = NT+(1:newNT);
8     p1 = elem(t,1); p2 = elem(t,2); p3 = elem(t,3);
9     p4 = edgeCutNumber(elem2edge(t,1));
10    elem(L,:) = [p4, p1, p2];
11    elem(R,:) = [p4, p3, p1];
12 end

```

这里把 `edgeCutNumber` 对应 `edge` 给出, 是为了方便与 `elem2edge(t,1)` 对应.

5.4.4 协调二分的程序实现

根据前面的说明, 当子三角形 peak 的对边在 base 集合中, 则该三角形要继续二分, 以去掉悬挂点. 为此, 我们要记录子三角形 peak 的对边, 即三角形的第一条边的自然序号. 显然只要存储在 `elem2edge(:,1)` 中即可, 于是有

```

1 elem2edge(L,1) = elem2edge(t,3);
2 elem2edge(R,1) = elem2edge(t,2);

```

这样, 对新的剖分执行

```
t = find(isCutEdge(elem2edge(:,1))==true);
```

就可找到需要的单元, 从而重复单元的记录即可.

```

1 % ----- Refine elements -----

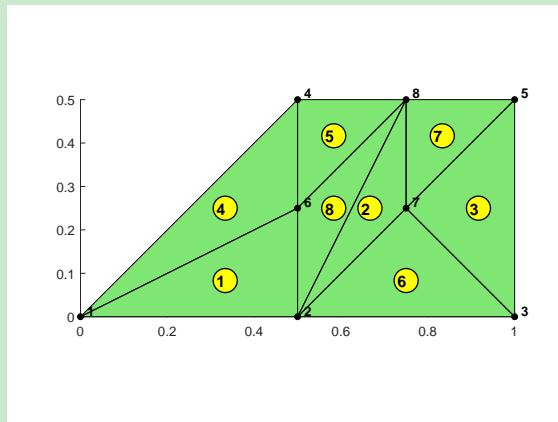
```

```

2 edgeCutNumber = zeros(NE,1);
3 edgeCutNumber(isCutEdge) = N + (1:Nbase);
4 for k = 1:2
5     t = find(isCutEdge(elem2edge(:,1))==true);
6     newNT = length(t);
7     if newNT==0, break; end
8     L = t; R = NT+(1:newNT);
9     p1 = elem(t,1); p2 = elem(t,2); p3 = elem(t,3);
10    p4 = edgeCutNumber(elem2edge(t,1));
11    elem(L,:) = [p4, p1, p2];
12    elem(R,:) = [p4, p3, p1];
13    elem2edge(L,1) = elem2edge(t,3);
14    elem2edge(R,1) = elem2edge(t,2);
15    NT = NT + newNT;
16 end

```

结果如下



5.4.5 Newest-node bisection 程序整理

CODE 5.2. bisect.m

```

1 function [node,elem] = bisect(node,elem,elemMarked)
2
3 % ----- Construct auxiliary data structure -----
4 aux = auxstructure(node,elem);
5 elem2edge = aux.elem2edge; edge = aux.edge; neighbor = aux.neighbor;
6 clear aux;
7 N = size(node,1); NT = size(elem,1); NE = size(edge,1);
8
9 % ----- (peak) base set -----

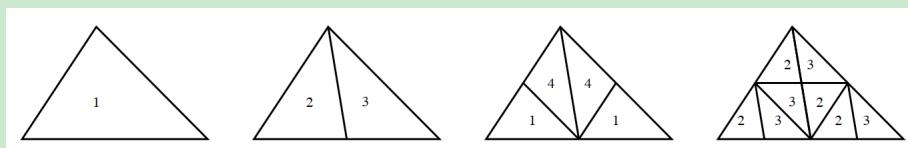
```

```

10 isCutEdge = false(NE,1);
11 while sum(elemMarked)>0
12     base = elem2edge(elemMarked,1);    isCutEdge(base) = true;
13     cc = NT*(base-1)+elemMarked;
14     refineNeighbor = unique(full(neighbor(cc)));
15     baseNeighbor = elem2edge(refineNeighbor,1);
16     elemMarked = refineNeighbor(~isCutEdge(baseNeighbor));
17 end
18
19 % ----- Add new nodes -----
20 Nbase = sum(isCutEdge);
21 edgeCutNumber = N+1:N+Nbase;
22 edgebase = edge(isCutEdge,:);
23 node(edgeCutNumber,:)= (node(edgebase(:,1),:)+...
24     node(edgebase(:,2),:))/2;
25
26 % ----- Refine elements -----
27 edgeCutNumber(isCutEdge) = (N+1:N+Nbase)';
28 for k = 1:2
29     t = find(isCutEdge(elem2edge(:,1))==true);
30     newNT = length(t);
31     if newNT==0, break; end
32     L = t; R = NT+(1:newNT);
33     p1 = elem(t,1); p2 = elem(t,2); p3 = elem(t,3);
34     p4 = edgeCutNumber(elem2edge(t,1));
35     elem(L,:)= [p4, p1, p2];
36     elem(R,:)= [p4, p3, p1];
37     elem2edge(L,1)= elem2edge(t,3);
38     elem2edge(R,1)= elem2edge(t,2);
39     NT = NT + newNT;
40 end

```

注 5.9 Newest-node bisection 只会产生四个相似类, 如下图所示



正因为如此, 该加密满足 shape regularity.

注 5.10 bisection.m 也可用来对网格进行全局加密, 只要把所有单元视为标记单元即可.

5.5 自适应有限元程序

主程序如下

```
1 clc;clear;close all;
2 % ----- Parameters -----
3 maxN = 1e4; theta = 0.4; maxIt = 30;
4 a1 = 0; b1 = 1; a2 = 0; b2 = 1;
5
6 % ----- Initial mesh and set up PDE data -----
7 Nx = 4; Ny = 4; h1 = 1/Nx; h2 = 1/Ny;
8 [node,elem] = squaremesh([a1 b1 a2 b2],h1,h2);
9 pde = pdedata();
10 % ----- Adaptive Finite Element Method -----
11 for k = 1:maxIt
12     % Step 1: SOLVE
13     bdFlag = setboundary(node,a1,b1,a2,b2);
14     u = Poisson(node,elem,bdFlag,pde);
15     figure(1); showmesh(node,elem);
16
17     % Step 2: ESTIMATE
18     eta = indicator(node,elem,u,pde);
19
20     % Step 3: MARK
21     elemMarked = mark(elem,eta,theta);
22
23     % Step 4: REFINE
24     [node,elem] = bisection(node,elem,elemMarked);
25
26     if (size(node,1)>maxN) || (k==maxIt)
27         bdFlag = setboundary(node,a1,b1,a2,b2);
28         u = Poisson(node,elem,bdFlag,pde);
29         break;
30     end
31
32 end
33
```

```

34 % ----- error analysis -----
35 ueexact = pde.ueexact;
36 ue = ueexact(node);
37 figure,
38 subplot(1,2,1), showsolution(node,elem,u);
39 zlabel('u');
40 subplot(1,2,2), showsolution(node,elem,ue);
41 zlabel('ue');
42 Eabs = u-ue; % Absolute errors
43 figure, showsolution(node,elem,Eabs); zlim('auto');

```

网格剖分如下

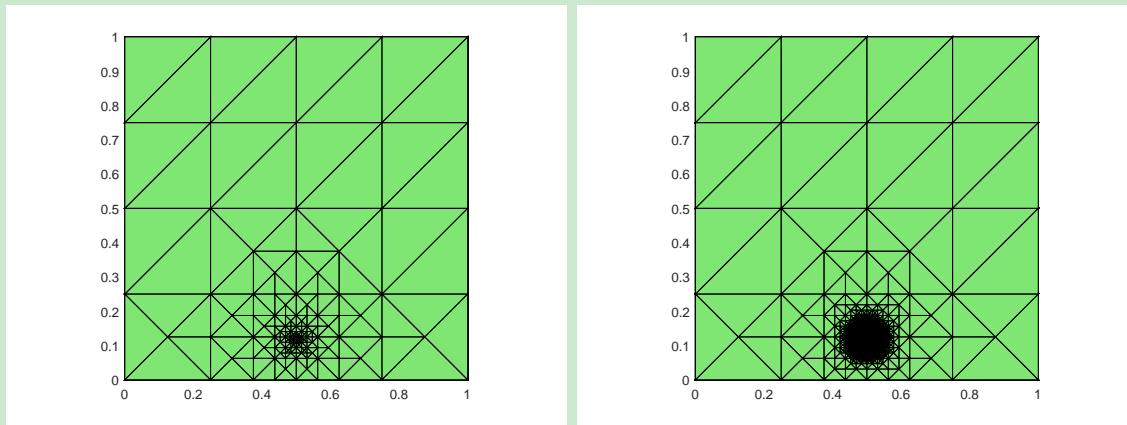


图 5.3. AFEM 的网格剖分

可以看到, 网格加密基本上在奇点附近.

迭代 30 次的数值解与精确解如下.

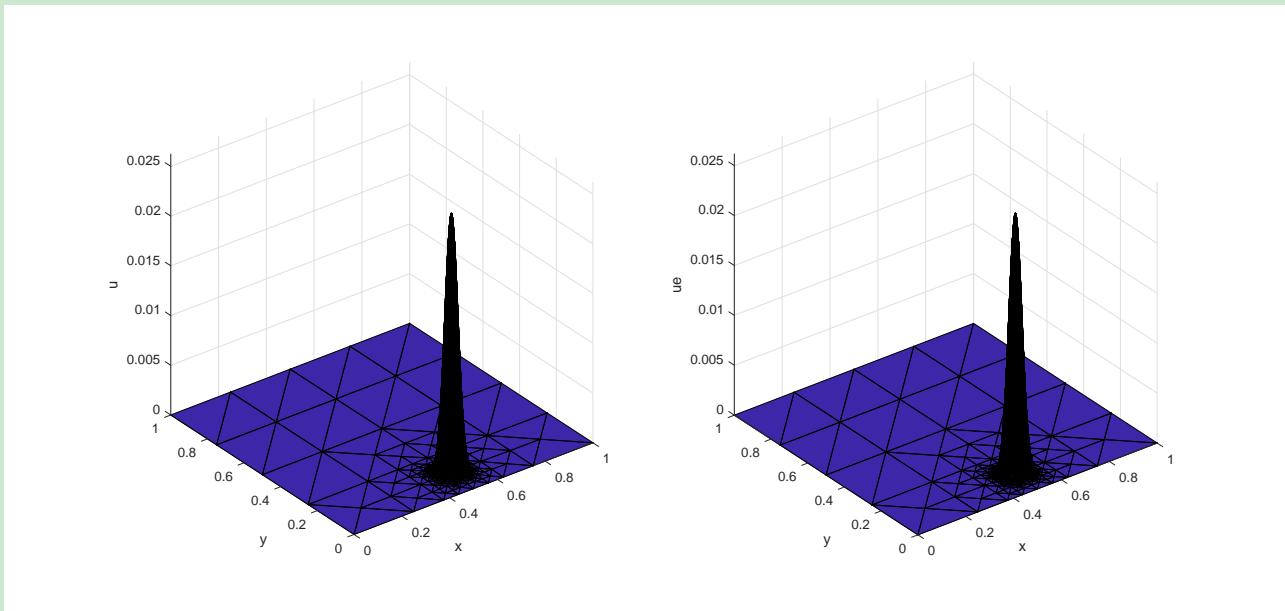


图 5.4. AFEM 的数值解与精确解

可以看到, 计算精度要比同时划分 50 份的高.

注 5.11 对二分法的网格剖分, 我们可使用多重网格法求解代数方程组. 这个方法比较复杂, 需要较长的时间讲清楚, 后面会介绍.

第六章 线弹性边值问题

本章考虑向量方程(也就是方程组), 即方程有多个未知函数, 经典的例子是线弹性边值问题.

6.1 线弹性边值问题简介

6.1.1 问题说明

设弹性体未受外力时所在区域为 $\Omega \subset \mathbb{R}^3$. 当它受体力 \mathbf{f} (在 Ω 中), 在 Ω 的一部分边界 Γ_1 受表面力 \mathbf{g} , 而在另一部分边界 Γ_0 上固定位移 $\mathbf{u} = \mathbf{0}$ 时, 弹性体就产生位移 \mathbf{u} . 在平衡状态下, 位移 \mathbf{u} 所满足的边值问题为

$$\begin{cases} -\partial_j \sigma_{ij}(\mathbf{u}) = f_i, & i = 1, 2, 3 \quad \text{in } \Omega, \\ \mathbf{u} = \mathbf{0} & \text{on } \Gamma_0, \\ \sigma_{ij}(\mathbf{u}) n_j = g_i, & i = 1, 2, 3 \quad \text{on } \Gamma_1, \end{cases} \quad (6.1)$$

这里约定: 凡在每一项中指标重复出现意味着从 1 到 3 (三维) 或 2 (二维) 求和. 上面的方程也可写为

$$\begin{cases} -\operatorname{div} \boldsymbol{\sigma} = \mathbf{f} & \text{in } \Omega, \\ \mathbf{u} = \mathbf{0} & \text{on } \Gamma_0, \\ \boldsymbol{\sigma} \mathbf{n} = \mathbf{g} & \text{on } \Gamma_1, \end{cases}$$

其中向量规定为列向量, 而 $\operatorname{div} \boldsymbol{\sigma}$ 是对矩阵 $\boldsymbol{\sigma}$ 的每行进行 (注意它是对称矩阵).

在以上式子中, 第一式为平衡方程, 其中的 σ_{ij} 为应力张量, 它与应变张量 $\varepsilon_{ij}(\mathbf{u})$ 满足如下的本构关系 (均匀的各项同性弹性体的 Hooke 定律)

$$\sigma_{ij}(\mathbf{u}) = \sigma_{ji}(\mathbf{u}) = \lambda \varepsilon_{kk}(\mathbf{u}) \delta_{ij} + 2\mu \varepsilon_{ij}(\mathbf{u}), \quad (6.2)$$

$$\varepsilon_{ij}(\mathbf{u}) = \varepsilon_{ji}(\mathbf{u}) = \frac{1}{2} (\partial_j u_i + \partial_i u_j), \quad (6.3)$$

而 λ 和 μ 为 Lamé 系数. 注意, 这里 $\varepsilon_{kk}(\mathbf{u})$ 按规定是对指标求和的, 显然有 (标量)

$$\varepsilon_{kk}(\mathbf{u}) = \partial_k u_k = \operatorname{div} \mathbf{u}.$$

这样, 平衡方程也可写为如下的紧凑形式

$$-\operatorname{div} (\lambda(\operatorname{div} \mathbf{u}) \mathbf{I} + 2\mu \boldsymbol{\varepsilon}(\mathbf{u})) = \mathbf{f} \quad \text{in } \Omega.$$

把 (6.2)-(6.3) 代入 (6.1), 可获得平衡方程的另一形式

$$-\mu \Delta \mathbf{u} - (\lambda + \mu) \operatorname{grad}(\operatorname{div} \mathbf{u}) = \mathbf{f} \quad \text{in } \Omega, \quad (6.4)$$

有时候会直接考虑该方程, 因为其中每个算子都是熟悉的.

6.1.2 连续变分问题

设

$$V = \left\{ \mathbf{v} \in H^1(\Omega)^3 : \mathbf{v} = \mathbf{0} \text{ on } \Gamma_0 \right\}, \quad (6.5)$$

令 $\mathbf{v} = (v_1, v_2, v_3)^T \in V$, 在 (6.1) 的平衡方程的两边乘以 v_i , 有

$$\int_{\Omega} -\partial_j \sigma_{ij}(\mathbf{u}) v_i dx = \int_{\Omega} f_i v_i dx,$$

这里遵循求和约定, 即上式实际上是求和. 分部积分有

$$-\int_{\partial\Omega} \sigma_{ij}(\mathbf{u}) v_i n_j ds + \int_{\Omega} \sigma_{ij}(\mathbf{u}) \partial_j v_i dx = \int_{\Omega} f_i v_i dx$$

或

$$-\int_{\partial\Omega} g_i v_i ds + \int_{\Omega} \sigma_{ij}(\mathbf{u}) \partial_j v_i dx = \int_{\Omega} f_i v_i dx.$$

由对称性,

$$\sigma_{ij}(\mathbf{u}) \partial_j v_i = \sigma_{ij}(\mathbf{u}) \frac{1}{2} (\partial_j v_i + \partial_i v_j) = \sigma_{ij}(\mathbf{u}) \varepsilon_{ij}(\mathbf{v}),$$

故

$$\int_{\Omega} \sigma_{ij}(\mathbf{u}) \varepsilon_{ij}(\mathbf{v}) dx = \int_{\Omega} \mathbf{f} \cdot \mathbf{v} dx + \int_{\Gamma_1} \mathbf{g} \cdot \mathbf{v} ds.$$

变分形式可写为三种形式.

1. 第一种形式 (elasticityVector) 为: 求 $\mathbf{u} \in V$ 使得,

$$a(\mathbf{u}, \mathbf{v}) = \ell(\mathbf{v}), \quad \mathbf{v} \in V, \quad (6.6)$$

式中,

$$a(\mathbf{u}, \mathbf{v}) = \int_{\Omega} \sigma_{ij}(\mathbf{u}) \varepsilon_{ij}(\mathbf{v}) dx, \quad \ell(\mathbf{v}) = \int_{\Omega} \mathbf{f} \cdot \mathbf{v} dx + \int_{\Gamma_1} \mathbf{g} \cdot \mathbf{v} ds.$$

文献中一般习惯引入如下记号

$$\mathbf{A} : \mathbf{B} = \sum_{ij} a_{ij} b_{ij}, \quad \mathbf{A} = (a_{ij}), \quad \mathbf{B} = (b_{ij}),$$

从而双线性形式可写为

$$a(\mathbf{u}, \mathbf{v}) = \int_{\Omega} \boldsymbol{\sigma}(\mathbf{u}) : \boldsymbol{\varepsilon}(\mathbf{v}) dx.$$

2. 第二种形式 (elasticityScalar). 注意到

$$\begin{aligned} \sigma_{ij}(\mathbf{u}) \varepsilon_{ij}(\mathbf{v}) &= (\lambda \varepsilon_{kk}(\mathbf{u}) \delta_{ij} + 2\mu \varepsilon_{ij}(\mathbf{u})) \varepsilon_{ij}(\mathbf{v}) \\ &= (\lambda \partial_k u_k \delta_{ij} + 2\mu \varepsilon_{ij}(\mathbf{u})) \varepsilon_{ij}(\mathbf{v}) \\ &= 2\mu \varepsilon_{ij}(\mathbf{u}) \varepsilon_{ij}(\mathbf{v}) + \lambda \partial_k u_k \varepsilon_{ii}(\mathbf{v}) \\ &= 2\mu \varepsilon_{ij}(\mathbf{u}) \varepsilon_{ij}(\mathbf{v}) + \lambda \partial_k u_k \partial_i u_i, \end{aligned}$$

双线性形式还可写为

$$\begin{aligned} a(\mathbf{u}, \mathbf{v}) &= 2\mu \int_{\Omega} \varepsilon_{ij}(\mathbf{u}) \varepsilon_{ij}(\mathbf{v}) dx + \lambda \int_{\Omega} \partial_i u_i \partial_j u_j dx, \\ &= 2\mu \int_{\Omega} \boldsymbol{\varepsilon}(\mathbf{u}) : \boldsymbol{\varepsilon}(\mathbf{v}) dx + \lambda \int_{\Omega} (\operatorname{div} \mathbf{u})(\operatorname{div} \mathbf{v}) dx. \end{aligned} \quad (6.7)$$

3. 第三种形式 (elasticityLaplace). 也可采用 (6.4) 的形式, 具体写出来即

$$\begin{cases} -\mu \Delta u_1 - (\lambda + \mu) \partial_x (\operatorname{div} \mathbf{u}) = f_1, \\ -\mu \Delta u_2 - (\lambda + \mu) \partial_y (\operatorname{div} \mathbf{u}) = f_2, \end{cases}$$

注意 $\operatorname{div} \mathbf{u}$ 是标量. 第一式乘以 v_1 , 并分部积分有

$$\begin{aligned} &\mu \left(\int_{\Omega} \nabla u_1 \cdot \nabla v_1 dx - \int_{\partial\Omega} \partial_n u_1 v_1 ds \right) \\ &- (\lambda + \mu) \left(\int_{\partial\Omega} (\operatorname{div} \mathbf{u}) v_1 n_x ds - \int_{\Omega} (\operatorname{div} \mathbf{u}) \partial_x v_1 dx \right) = \int_{\Omega} f_1 v_1 dx. \end{aligned}$$

类似可获得第二个式子对应的结果, 将它们相加有

$$\begin{aligned} &\mu \int_{\Omega} \nabla \mathbf{u} \cdot \nabla \mathbf{v} dx + (\lambda + \mu) \int_{\Omega} (\operatorname{div} \mathbf{u})(\operatorname{div} \mathbf{v}) dx \\ &- \mu \int_{\partial\Omega} \partial_n \mathbf{u} \cdot \mathbf{v} ds - (\lambda + \mu) \int_{\partial\Omega} (\operatorname{div} \mathbf{u})(\mathbf{v} \cdot \mathbf{n}) ds = \int_{\Omega} \mathbf{f} \cdot \mathbf{v} dx. \end{aligned} \quad (6.8)$$

iFEM 给出了该变分形式 Dirichlet 问题的程序 (elasticity.m).

6.1.3 有限元方法

以下考虑 $\Gamma_1 = \emptyset$ 的情形. 协调一次元空间为

$$V_h = \left\{ \mathbf{v} \in (H_0^1(\Omega))^2 : \mathbf{v}|_K \in (\mathbb{P}_1(K))^2, K \in \mathcal{T}_h \right\},$$

相应的有限元问题为: 求 $\mathbf{u}_h \in V_h$ 使得

$$a(\mathbf{u}_h, \mathbf{v}) = \ell(\mathbf{v}), \quad \mathbf{v} \in V_h.$$

定理 6.1 设 \mathbf{u} 和 \mathbf{u}_h 分别为连续变分问题和近似变分问题的解, 则

$$\|\mathbf{u} - \mathbf{u}_h\|_{1,\Omega} \lesssim (2\mu + \lambda)h|\mathbf{u}|_{2,\Omega}.$$

注 6.1 对几乎不可压缩的材料, $\lambda \gg \mu$. 从上面的误差估计以及数值实例可以看到, 当 $\lambda \rightarrow \infty$ 时, 协调一次元方法不再收敛, 称为闭锁 (locking) 现象. 事实上, 我们的确可以构造出不收敛的例子.

6.2 刚度矩阵与载荷向量的装配

对向量方程, 因含有两个未知函数, 装配的过程与它们的排列顺序相关. 为了方便, 考虑如下变分形式

$$a(\mathbf{u}, \mathbf{v}) = \ell(\mathbf{v}),$$

式中,

$$a(\mathbf{u}, \mathbf{v}) = \sum_{i,j=1}^2 \int_{\Omega} u_i v_j dx, \quad \ell(\mathbf{v}) = \sum_{i=1}^2 \int_{\Omega} f_i v_i dx.$$

6.2.1 单元的向量法分析

对 $\mathbf{u} = [u_1, u_2]^T$, 为了避免下标的混乱, 我们记 $\bar{u} = u_1$ 和 $\underline{u} = u_2$, 相应的节点基展开为

$$\bar{u} = \bar{u}_1 \varphi_1 + \cdots + \bar{u}_n \varphi_n = \Psi \bar{U}, \quad \underline{u} = \underline{u}_1 \varphi_1 + \cdots + \underline{u}_n \varphi_n = \Psi \underline{U},$$

其中,

$$\Psi = [\varphi_1, \dots, \varphi_n]^T, \quad \bar{U} = [\bar{u}_1, \dots, \bar{u}_n]^T, \quad \underline{U} = [\underline{u}_1, \dots, \underline{u}_n]^T.$$

于是,

$$\mathbf{u} = \sum_{i=1}^n \bar{u}_i \bar{\varphi}_i + \sum_{i=1}^n \underline{u}_i \underline{\varphi}_i, \quad (6.9)$$

式中,

$$\bar{\varphi}_j = \begin{bmatrix} \varphi_j \\ 0 \end{bmatrix}, \quad \underline{\varphi}_j = \begin{bmatrix} 0 \\ \varphi_j \end{bmatrix}, \quad j = 1, \dots, n.$$

令

$$N_j = [\bar{\varphi}_j, \underline{\varphi}_j] = \begin{bmatrix} \bar{\varphi}_j^T \\ \underline{\varphi}_j^T \end{bmatrix}, \quad j = 1, \dots, n,$$

它是对称的, 对向量法编程, 通常按行考虑 (符合矩阵向量运算). 显然有

$$\mathbf{u} = \sum_{i=1}^n N_i U_i, \quad U_i = \begin{bmatrix} \bar{u}_i \\ \underline{u}_i \end{bmatrix}$$

或

$$\mathbf{u} = \begin{bmatrix} \bar{u} \\ \underline{u} \end{bmatrix} = N U,$$

式中,

$$N = [N_1, \dots, N_n], \quad U = [U_1; \dots; U_n],$$

这里向量之间的分号表示按列拉直, 与 MATLAB 的运算一致.

这样, 双线性形式和右端可写为

$$a(\mathbf{u}, \mathbf{v}) = \int_{\Omega} \mathbf{v}^T \mathbf{u} dx = V^T \int_{\Omega} N^T N dx U = V^T A U, \quad A = \int_{\Omega} N^T N dx,$$

$$\ell(\mathbf{v}) = \int_{\Omega} \mathbf{v}^T \mathbf{f} dx = V^T \int_{\Omega} N^T \mathbf{f} dx = V^T F, \quad F = \int_{\Omega} N^T \mathbf{f} dx,$$

于是

$$AU = F.$$

可以看到, 在把向量 $U_i = \begin{bmatrix} \bar{u}_i \\ u_i \end{bmatrix}$ 视为变量的情况下, 分析过程与标量情形完全相同.

事实上, 对三角形单元 β , 设三个顶点分别为 $\delta_1, \delta_2, \delta_3$, 取 φ_i 为局部基函数, 同上有

$$A_{\beta} U_{\beta} = F_{\beta},$$

式中,

$$A_{\beta} = \int_{\beta} N^T N dx, \quad F_{\beta} = \int_{\beta} N^T \mathbf{f} dx,$$

$$N = [N_1, N_2, N_3], \quad U_{\beta} = [U_1; U_2; U_3].$$

单元矩阵和载荷向量可分块为

$$A_{\beta} = (K_{st})_{3 \times 3}, \quad K_{ij} = \int_{\beta} N_s^T N_t dx \in \mathbb{R}^{2 \times 2},$$

$$F_{\beta} = (F_s)_{3 \times 1}, \quad F_s = \int_{\beta} N_s^T \mathbf{f} dx \in \mathbb{R}^2.$$

此时, K_{ij} 和 F_i 相当于标量情形的 k_{ij} 和 f_i .

6.2.2 sparse 装配指标

把单元刚度矩阵按分量记为 $A_{\beta} = (k_{ij})_{6 \times 6}$, 可以给出相应的 `sparse` 装配指标. 即首先给出 k_{11} 所有单元的 (i, j, s) , 并记为 i_{11}, j_{11}, s_{11} . 接着按单元矩阵的行给出其他分量的, 排列为

$$\begin{bmatrix} i_{11} & j_{11} & s_{11} \\ i_{12} & j_{12} & s_{12} \\ \vdots & \vdots & \vdots \\ i_{66} & j_{66} & s_{66} \end{bmatrix}.$$

笔者开始也这样考虑过, 但相较于按分量进行分块的方式, 它不利于 MATLAB 的向量运算 (或者说为了用向量运算, 一些数据需要稍复杂的存储).

若把变量按标量编程法排列, 即

$$a_\beta(\mathbf{u}, \mathbf{v}) \rightarrow \begin{bmatrix} A_{11}^\beta & A_{12}^\beta \\ A_{21}^\beta & A_{22}^\beta \end{bmatrix} \begin{bmatrix} \bar{U}_\beta \\ U_\beta \end{bmatrix},$$

则容易发现

$$A_{ij}^\beta = (K_{st}(i, j))_{3 \times 3}, \quad 1 \leq i, j \leq 2,$$

即把每个 K_{st} 的 (i, j) 位置元素拿出来组成的矩阵恰是分块矩阵的 A_{ij}^β . 本文称这种分块的方式为标量法编程.

1. 单元分析也可按分量进行分析, 但用向量分析法更加清楚.
2. 分块单元刚度矩阵的向量稀疏指标也容易获得. 若直接用标量法编程, 则有

$$\bar{U}_\beta \leftrightarrow \text{elem}(:), \quad U_\beta \leftrightarrow \text{elem}(:) + N.$$

显然向量法只需要改为

$$\bar{U}_\beta \leftrightarrow 2 * \text{elem}(:) - 1, \quad U_\beta \leftrightarrow 2 * \text{elem}(:).$$

本文只采用前者.

刚度矩阵

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

的每个块都可直接按照标量情形的方法进行装配, 即用 `sparse(ii, jj, ss, N, N)` 生成, 其中的 `ii, jj` 是通用的, 由 CODE. 4.5 给出. 而 `ss` 如下获得: 将单元刚度矩阵行拉直, 逐个单元拼成一个矩阵(每行对应一个单元的拉直向量), 然后再拉直为一个列向量. 若把这些块给出的拉直向量分别记为 `ss11, ss12, ss21, ss22`, 则可如下装配

```

1 A11 = sparse(ii, jj, ss11, N, N); A12 = sparse(ii, jj, ss12, N, N);
2 A21 = sparse(ii, jj, ss21, N, N); A22 = sparse(ii, jj, ss22, N, N);
3 A = [A11, A12; A21, A22];

```

为了避免对稀疏矩阵进行运算, 上面分块装配可如下改进

CODE 6.1. 向量方程的 `sparse` 装配指标

```

1 ii11 = ii;    jj11 = jj;    ii12 = ii;    jj12 = jj+N;
2 ii21 = ii+N;  jj21 = jj;    ii22 = ii+N;  jj22 = jj+N;
3 ii = [ii11; ii12; ii21; ii22];

```

```

4 jj = [jj11; jj12; jj21; jj22];
5 ss = [ss11; ss12; ss21; ss22];
6 A = sparse(ii,jj,ss,2*N,2*N);

```

单元载荷 F_β 分成三块, 每块 F_s 都是两行的向量. 类似刚度矩阵,

$$F_i^\beta = (F_s(i))_{3 \times 1}, \quad i = 1, 2,$$

即把每个 F_s 的第 i 个元素拿出来组成的向量恰是分块情形的 F_i^β . 给定每块的向量 $F1, F2$, 则如下装配

```

1 ff = accumarray([elem(:); elem(:)+N], [F1(:); F2(:)], [2*N 1]);

```

6.2.3 双线性分量的配对

我们用系统方程分析法来考察一下分量配对 (v_i, u_j) 对应的单元矩阵和单元向量, 从而能够明确装配的具体对应. 对 \mathbf{u} 进行分块形式的展开 (即式 (6.9)), 变分形式对应如下若干个方程

$$\begin{cases} a(\mathbf{u}, \mathbf{v}) = \ell(\mathbf{v}), & \mathbf{v} = \bar{\varphi}_j, \quad j = 1, \dots, N; \\ a(\mathbf{u}, \mathbf{v}) = \ell(\mathbf{v}), & \mathbf{v} = \underline{\varphi}_j, \quad j = 1, \dots, N. \end{cases}$$

注意到 $\bar{\varphi}_j = [\varphi_j, 0]^T$, 第一行方程只会留下检验函数 $\bar{v} = \varphi_j$ 对应的项, 从而

$$a(\mathbf{u}, \mathbf{v}) = \int_{\Omega} \bar{u} \bar{v} dx + \int_{\Omega} \underline{u} \bar{v} dx, \quad \bar{v} = \varphi_j,$$

即

$$a(\mathbf{u}, \mathbf{v}) = \sum_{i=1}^N \int_{\Omega} \varphi_j \varphi_i dx \bar{u}_i + \sum_{i=1}^N \int_{\Omega} \varphi_j \varphi_i dx \underline{u}_i,$$

从而第一行双线性形式对应

$$a(\mathbf{u}, \mathbf{v}), \quad \mathbf{v} = \bar{\varphi}_j, \quad j = 1, \dots, N \quad \rightarrow \quad A_{11} \bar{U} + A_{12} \underline{U} = [A_{11}, A_{12}] \begin{bmatrix} \bar{U} \\ \underline{U} \end{bmatrix},$$

这里 $A_{11} = (\varphi_j, \varphi_i)$ 对应 (\bar{v}, \bar{u}) , 而 $A_{12} = (\varphi_j, \varphi_i)$ 对应 (\bar{v}, \underline{u}) . 类似地, 第二行方程对应

$$a(\mathbf{u}, \mathbf{v}), \quad \mathbf{v} = \underline{\varphi}_j, \quad j = 1, \dots, N \quad \rightarrow \quad A_{21} \bar{U} + A_{22} \underline{U} = [A_{21}, A_{22}] \begin{bmatrix} \bar{U} \\ \underline{U} \end{bmatrix},$$

这里 A_{21} 对应 (\underline{v}, \bar{u}) , 而 A_{22} 对应 $(\underline{v}, \underline{u})$. 两行方程组拼接起来就是

$$a(\mathbf{u}, \mathbf{v}) \rightarrow \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} \bar{U} \\ \underline{U} \end{bmatrix}.$$

从上面的分析可以看到,

$$(v_1, u_1) \rightarrow \begin{bmatrix} A_{11} & O \\ O & O \end{bmatrix} \begin{bmatrix} \bar{U} \\ U \end{bmatrix}; \quad (v_1, u_2) \rightarrow \begin{bmatrix} O & A_{12} \\ O & O \end{bmatrix} \begin{bmatrix} \bar{U} \\ U \end{bmatrix};$$

$$(v_2, u_1) \rightarrow \begin{bmatrix} O & O \\ A_{21} & O \end{bmatrix} \begin{bmatrix} \bar{U} \\ U \end{bmatrix}; \quad (v_2, u_2) \rightarrow \begin{bmatrix} O & O \\ O & A_{22} \end{bmatrix} \begin{bmatrix} \bar{U} \\ U \end{bmatrix}.$$

而 (v_i, u_j) 对应的 A_{ij} 恰是单个函数情形给出的刚度矩阵. 为此, 刚度矩阵的装配算法如下

算法 1 向量方程刚度矩阵的装配

1. 设 $\mathbf{u} = [u_1, \dots, u_d]^T$, $\mathbf{v} = [v_1, \dots, v_d]^T$ (这里 $d = 2$), 而 U_i 是 u_i 在节点基下的展开向量, 则刚度矩阵形如

$$\begin{bmatrix} A_{11} & \cdots & A_{1d} \\ \vdots & \ddots & \vdots \\ A_{d1} & \cdots & A_{dd} \end{bmatrix} \leftrightarrow \begin{bmatrix} U_1 \\ \vdots \\ U_d \end{bmatrix}.$$

2. A_{ij} 是对应 (v_i, u_j) 的双线性形式给出的刚度矩阵 (标量情形), 按前面介绍的装配算法实施.
-

类似给出载荷向量的装配, 右端为

$$\ell(\mathbf{v}) = \int_{\Omega} \mathbf{f} \cdot \mathbf{v} dx = \sum_{i=1}^d \int_{\Omega} f_i v_i dx.$$

算法 2 向量方程刚度矩阵的装配

1. 设 $\mathbf{f} = [f_1, \dots, f_d]^T$, 而 U_i 是 u_i 在节点基下的展开向量, 则载荷向量形如

$$\begin{bmatrix} F_1 \\ \vdots \\ F_d \end{bmatrix} \leftrightarrow \begin{bmatrix} U_1 \\ \vdots \\ U_d \end{bmatrix}.$$

2. F_i 是对应 (v_i, f_i) 的线性形式给出的载荷向量 (标量情形), 按前面介绍的装配算法实施.
-

6.3 第一种形式的变分问题

6.3.1 刚度矩阵的计算

命

$$\sigma = \begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{12} \end{bmatrix}, \quad \varepsilon = \begin{bmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ 2\varepsilon_{12} \end{bmatrix},$$

则

$$\int_{\beta} \sigma_{ij}(\mathbf{u}) \varepsilon_{ij}(\mathbf{v}) dx = \int_{\beta} \varepsilon^T(\mathbf{v}) \sigma(\mathbf{u}) dx,$$

且有 Hooke 定律

$$\sigma = R\varepsilon, \quad R = \begin{bmatrix} \lambda + 2\mu & \lambda & 0 \\ \lambda & \lambda + 2\mu & 0 \\ 0 & 0 & \mu \end{bmatrix}.$$

设 β 对应的节点基为 ϕ_1, ϕ_2, ϕ_3 , 记

$$U_{\beta} = [\bar{u}_1, \underline{u}_1, \bar{u}_2, \underline{u}_2, \bar{u}_3, \underline{u}_3]^T,$$

$$N = [N_1, N_2, N_3], \quad N_i = [\bar{\phi}_i, \underline{\phi}_i] = \phi_i \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},$$

则在单元 β 上有

$$\mathbf{u} = NU_{\beta}, \quad \mathbf{v} = NV_{\beta}.$$

此时应变向量近似为

$$\varepsilon = \begin{bmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ 2\varepsilon_{12} \end{bmatrix} = \begin{bmatrix} \partial_1 \bar{u} \\ \partial_2 \underline{u} \\ \partial_2 \bar{u} + \partial_1 \underline{u} \end{bmatrix} = \begin{bmatrix} \partial_1 & 0 \\ 0 & \partial_2 \\ \partial_2 & \partial_1 \end{bmatrix} \begin{bmatrix} \bar{u} \\ \underline{u} \end{bmatrix} \approx \begin{bmatrix} \partial_1 & 0 \\ 0 & \partial_2 \\ \partial_2 & \partial_1 \end{bmatrix} NU_{\beta} =: BU_{\beta},$$

式中,

$$B = [B_1, B_2, B_3], \quad B_i = \begin{bmatrix} \partial_1 \phi_i & 0 \\ 0 & \partial_2 \phi_i \\ \partial_2 \phi_i & \partial_1 \phi_i \end{bmatrix}.$$

于是双线性形式为

$$\int_{\beta} \sigma_{ij}(\mathbf{u}) \varepsilon_{ij}(\mathbf{v}) dx = V_{\beta}^T \int_{\beta} B^T R B dx U_{\beta} =: V_{\beta}^T K_{\beta} U_{\beta},$$

这里

$$K_\beta = \int_{\beta} B^T R B dx = (K_{st})_{3 \times 3}, \quad K_{st} = \int_{\beta} B_s^T R B_t dx \in \mathbb{R}^{2 \times 2}.$$

设

$$K_{st} = (k_{ij}^{st})_{2 \times 2},$$

则有

$$\begin{cases} k_{11}^{st} = \int_{\beta} (\lambda + 2\mu) \partial_1 \phi_s \partial_1 \phi_t + \mu \partial_2 \phi_s \partial_2 \phi_t, \\ k_{12}^{st} = \int_{\beta} \lambda \partial_1 \phi_s \partial_2 \phi_t + \mu \partial_2 \phi_s \partial_1 \phi_t, \\ k_{21}^{st} = \int_{\beta} \lambda \partial_2 \phi_s \partial_1 \phi_t + \mu \partial_1 \phi_s \partial_2 \phi_t, \\ k_{22}^{st} = \int_{\beta} (\lambda + 2\mu) \partial_2 \phi_s \partial_2 \phi_t + \mu \partial_1 \phi_s \partial_1 \phi_t. \end{cases}$$

下面说明一下程序的实现.

- 注意到 $\partial_j \phi_s$ 是常数, 我们有

$$(\partial_i \phi_s, \partial_j \phi_t)_\beta = \partial_i \phi_s \cdot \partial_j \phi_t \cdot |\beta|,$$

从而要计算局部节点基, 也就是面积坐标函数的导数, 以及单元的面积, 这些计算在前面已经说明过, 例如节 5.2.2. 计算中用三维数组 Dphi 存储三个基函数的导数, 如下

```

1 function [Dphi,area] = gradbasis(node,elem)
2
3 z1 = node(elem(:,1),:);
4 z2 = node(elem(:,2),:);
5 z3 = node(elem(:,3),:);
6 e1 = z2-z3; e2 = z3-z1; e3 = z1-z2;
7 area = 0.5*(-e3(:,1).*e2(:,2)+e3(:,2).*e2(:,1));
8
9 grad1 = [e1(:,2), -e1(:,1)]./(2*area); % stored in rows
10 grad2 = [e2(:,2), -e2(:,1)]./(2*area);
11 grad3 = -(grad1+grad2);
12
13 NT = size(elem,1);
14 Dphi(1:NT,:,:1) = grad1;
15 Dphi(1:NT,:,:2) = grad2;
16 Dphi(1:NT,:,:3) = grad3;

```

这里, grad1 的行对应单元.

- 可事先算出所有单元的 $(\partial_i \phi_s, \partial_j \phi_t)_\beta$, 我们用元胞数组 `Dbase` 存储, 它是 2×2 的, 每块存储一个导数配对. 根据装配的说明, 单元矩阵按行拉直, 然后逐个单元拼在一起. 为此这里每块也遵循这个规定. 程序如下

```

1 Dbase = cell(2,2);
2 for i = 1:2
3     for j = 1:2
4         k11 = Dphi(:,i,1).*Dphi(:,j,1).*area;
5         k12 = Dphi(:,i,1).*Dphi(:,j,2).*area;
6         k13 = Dphi(:,i,1).*Dphi(:,j,3).*area;
7         k21 = Dphi(:,i,2).*Dphi(:,j,1).*area;
8         k22 = Dphi(:,i,2).*Dphi(:,j,2).*area;
9         k23 = Dphi(:,i,2).*Dphi(:,j,3).*area;
10        k31 = Dphi(:,i,3).*Dphi(:,j,1).*area;
11        k32 = Dphi(:,i,3).*Dphi(:,j,2).*area;
12        k33 = Dphi(:,i,3).*Dphi(:,j,3).*area;
13        K = [k11,k12,k13,k21,k22,k23,k31,k32,k33]; % stored in ...
14        rows
15    end
16 end

```

- 单元刚度矩阵如下装配

```

1 % ----- Assemble stiffness matrix -----
2 ss11 = (lambda+2*mu)*Dbase{1,1} + mu*Dbase{2,2};
3 ss12 = lambda*Dbase{1,2} + mu*Dbase{2,1};
4 ss21 = lambda*Dbase{2,1} + mu*Dbase{1,2};
5 ss22 = (lambda+2*mu)*Dbase{2,2} + mu*Dbase{1,1};
6 ii = [ii11; ii12; ii21; ii22];
7 jj = [jj11; jj12; jj21; jj22];
8 ss = [ss11; ss12; ss21; ss22];
9 kk = sparse(ii,jj,ss,2*N,2*N);

```

这里的装配指标由 (6.1) 给出.

6.3.2 载荷向量的计算

右端为

$$\int_{\beta} \mathbf{f} \cdot \mathbf{v} dx = \int_{\beta} \mathbf{v}^T \mathbf{f} dx = V_{\beta}^T \int_{\beta} N^T \mathbf{f} dx = V_{\beta}^T F_{\beta},$$

单元载荷向量为

$$F_\beta = \int_{\beta} N^T \mathbf{f} dx = (F_i)_{3 \times 1},$$

式中,

$$F_i = \int_{\beta} N_i^T \mathbf{f} dx = \int_{\beta} \begin{bmatrix} \phi_i & 0 \\ 0 & \phi_i \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} dx = \int_{\beta} \begin{bmatrix} f_1 \phi_i \\ f_2 \phi_i \end{bmatrix} dx.$$

我们先计算第一个分量所有单元的结果, 再计算第二个分量的, 即分别考虑

$$f_i \phi_1, f_i \phi_2, f_i \phi_3, i = 1, 2,$$

这对应的标量情形的载荷向量, 与常规计算一致.

中点型积分公式如下计算

```

1 % mid-point quadrature rule
2 x1 = node(elem(:,1),1); y1 = node(elem(:,1),2);
3 x2 = node(elem(:,2),1); y2 = node(elem(:,2),2);
4 x3 = node(elem(:,3),1); y3 = node(elem(:,3),2);
5 xc = 1/3*(x1+x2+x3); yc = 1/3*(y1+y2+y3); pc = [xc,yc];
6
7 f1 = f(pc).*area./3; f2 = f1; f3 = f1;
8 F1 = [f1(:,1),f2(:,1),f3(:,1)];
9 F2 = [f1(:,2),f2(:,2),f3(:,2)];
10 ff = accumarray([elem(:); elem(:)+N], [F1(:);F2(:)], [2*N 1]);

```

也可用前面介绍的三角形上的 Gauss 求积公式计算, 注意此时的被积函数为 (单元排成一行)

$$f_i \phi_1, f_i \phi_2, f_i \phi_3, i = 1, 2.$$

命令 `[lambda, weight] = quadpts(n)` 给出权重 `weight`, 它是一个向量; 而 `lambda` 是重心坐标函数的相应函数值, 共有三列, 列对应 $\lambda_j = \phi_j$. 这样, $f\phi_j$ 对应的函数值为 `f(xy).*lambda(:,j)`, 其中 `xy` 为重心坐标对应的直角坐标. 对 $\mathbf{f} = [f_1, f_2]$, 可如下计算

```

1 % Gauss quadrature rule
2 [lambda, weight] = quadpts(2);
3 f1 = zeros(NT,2); f2 = f1; f3 = f1;
4 for iel = 1:NT
5     vK = node(elem(iel,:)); % vertices of K
6     xy = lambda*vK; fxy = f(xy); % fxy = [f1xy, f2xy]
7     fv1 = fxy.*[lambda(:,1),lambda(:,1)]; % (f, phi1)
8     fv2 = fxy.*[lambda(:,2),lambda(:,2)]; % (f, phi2)
9     fv3 = fxy.*[lambda(:,3),lambda(:,3)]; % (f, phi3)

```

```

10
11 f1(iel,:) = area(iel)*weight*f1;
12 f2(iel,:) = area(iel)*weight*f2;
13 f3(iel,:) = area(iel)*weight*f3;
14 end
15 F1 = [f1(:,1),f2(:,1),f3(:,1)];
16 F2 = [f1(:,2),f2(:,2),f3(:,2)];
17 ff = accumarray([elem(:); elem(:)+N], [F1(:); F2(:)], [2*N 1]);

```

注意 weight 是行向量.

6.3.3 Neumann 边界条件

在 Γ_1 上有附加载荷, 所得边界条件为

$$\sigma \mathbf{n} = \mathbf{g} \quad \text{on } \Gamma_1,$$

不妨称为 Neumann 边界条件. 变分形式的右端要加上

$$\int_{\Gamma_1} \mathbf{g} \cdot \mathbf{v} ds.$$

设 $\gamma = (\delta_1, \delta_2)$ 是边界边, 相应的局部节点基为 $\tilde{\phi}_1, \tilde{\phi}_2$. 记

$$U_\gamma = [\bar{u}_1, \underline{u}_1, \bar{u}_2, \underline{u}_2],$$

$$\tilde{N} = [\tilde{N}_1, \tilde{N}_2], \quad \tilde{N}_i = \tilde{\phi}_i \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix},$$

则

$$\int_{\gamma} \mathbf{g} \cdot \mathbf{v} ds = \int_{\gamma} \mathbf{v}^T \mathbf{g} ds = V_\gamma^T \int_{\gamma} \tilde{N}^T \mathbf{g} ds =: V_\gamma^T F_\gamma,$$

式中,

$$F_\gamma = \int_{\gamma} \tilde{N}^T \mathbf{g} ds = [F_1, F_2], \quad F_i = \int_{\gamma} \begin{bmatrix} g_1 \tilde{\phi}_i \\ g_2 \tilde{\phi}_i \end{bmatrix} ds.$$

由梯形公式, 有

$$F_i = \int_{\gamma} \mathbf{g} \tilde{\phi}_i ds = \int_{\gamma} \sigma \mathbf{n} \tilde{\phi}_i ds = \frac{h_\gamma}{2} ((\sigma \mathbf{n} \tilde{\phi}_i)(\delta_1) + (\sigma \mathbf{n} \tilde{\phi}_i)(\delta_2)) = \frac{h_\gamma}{2} (\sigma \mathbf{n})(\delta_i),$$

其中

$$h_\gamma = |\delta_2 - \delta_1| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}.$$

注意到

$$h_\gamma \sigma \mathbf{n} = \sigma(h_\gamma \mathbf{n}),$$

而 $h_\gamma \mathbf{n}$ 恰是定向边 γ 顺时针旋转 90° 或 $-\gamma$ 逆时针旋转 90° 所得. 这样, 设 $-\gamma$ 的坐标为 (a, b) , 则 $h_\gamma \mathbf{n}$ 的坐标为 $(-b, a)$.

类似载荷向量的装配, 先计算所有 F_i 的第一个分量并放在一起, 再计算第二个分量的放在一起. 我们在 elasticitydata.m 中定义

$$g_N = [\sigma_{11}(x, y), \sigma_{22}(x, y), \sigma_{12}(x, y)],$$

结合一维问题的装配算法, Neumann 边界条件可如下编程

```

1 % ----- Neumann boundary condition -----
2 elemN = bdStruct.elemN;
3 if ~isempty(elemN)
4     g_N = pde.g_N;
5     z1 = node(elemN(:,1),:); z2 = node(elemN(:,2),:);
6     e = z1-z2; % e = z2-z1
7     ne = [-e(:,2),e(:,1)]; % scaled ne
8     Sig1 = g_N(z1); Sig2 = g_N(z2);
9     F11 = sum(ne.*Sig1(:,[1,3]),2)./2;
10    F12 = sum(ne.*Sig2(:,[1,3]),2)./2;
11    F21 = sum(ne.*Sig1(:,[3,2]),2)./2;
12    F22 = sum(ne.*Sig2(:,[3,2]),2)./2;
13    FN = [F11,F12,F21,F22];
14    ff = ff + accumarray([elemN(:); elemN(:)+N], FN(:), [2*N 1]);
15 end

```

6.3.4 Dirichlet 边界条件

边界定义函数见前面介绍的 setboundary.m. Dirichlet 边界条件类似标量情形处理, 如下

```

1 % ----- Dirichlet boundary condition -----
2 g_D = pde.g_D; eD = bdStruct.eD;
3 isBdNode = false(N,1); isBdNode(eD) = true;
4 bdNode = find(isBdNode); freeNode = find(~isBdNode);
5 pD = node(bdNode,:);
6 bdDof = [bdNode; bdNode+N]; freeDof = [freeNode; freeNode+N];
7 u = zeros(2*N,1); uD = g_D(pD); u(bdDof) = uD(:);
8 ff = ff - kk*u;
9 % ----- Solver -----
10 u(freeDof) = kk(freeDof,freeDof)\ff(freeDof);

```

6.3.5 程序整理

设左右边界是 Neumann 边界条件, 主程序如下

CODE 6.2. main_elasticity.m

```
1 clc;clear;close all
2 % ----- Mesh and boudary conditions -----
3 a1 = 0; b1 = 1; a2 = 0; b2 = 1;
4 Nx = 10; Ny = 10; h1 = (b1-a1)/Nx; h2 = (b2-a2)/Ny;
5 [node,elem] = squaremesh([a1 b1 a2 b2],h1,h2);
6
7 bdNeumann = 'abs(x-0)<1e-4 | abs(x-1)<1e-4'; % string for Neumann
8 bdStruct = setboundary(node,elem,bdNeumann);
9
10 % ----- PDE data -----
11 lambda = 1; mu = 1;
12 para.lambda = lambda; para.mu = mu;
13 pde = elasticitydata(para);
14
15 % ----- elasticity -----
16 u = elasticity(node,elem,pde,bdStruct);
17
18 % ----- error analysis -----
19 u = reshape(u,[],2);
20 ueexact = pde.ueexact; ue = ueexact(node);
21 id = 1;
22 figure,
23 subplot(1,2,1), showsolution(node,elem,u(:,id));
24 zlabel('u');
25 subplot(1,2,2), showsolution(node,elem,ue(:,id));
26 zlabel('ue');
27 Eabs = u-ue; % Absolute errors
28 figure, showsolution(node,elem,Eabs(:,id)); zlim('auto');
29 format shorte
30 Err = norm(Eabs)./norm(ue)
```

结果为

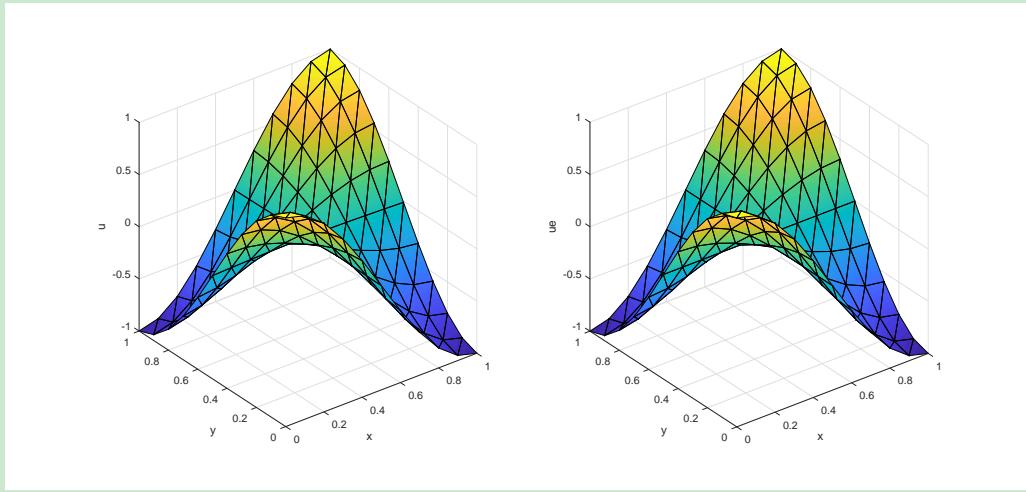


图 6.1. 变分形式 (6.6) 的数值解与精确解

函数文件为

CODE 6.3. elasticity.m

```

1 function u = elasticity(node, elem, pde, bdStruct)
2 %ElasticityVector Conforming P1 FEM of linear elasticity equation
3 %
4 %      u = [u1, u2]
5 %      -div (sigma) = f in \Omega
6 %      Dirichlet boundary condition u = [g1_D, g2_D] on \Gamma_D
7 %      Neumann boundary condition \sigma*n = g on \Gamma_N
8 %      \sigma = (sigma_{ij}): stress tensor, 1≤i,j≤2
9
10 N = size(node,1); NT = size(elem,1); Ndof = 3;
11 mu = pde.mu; lambda = pde.lambda; f = pde.f;
12 % ----- Compute (Dbase,Djbase) -----
13 [Dphi,area] = gradbasis(node,elem);
14 Dbase = cell(2,2);
15 for i = 1:2
16     for j = 1:2
17         k11 = Dphi(:,i,1).*Dphi(:,j,1).*area;
18         k12 = Dphi(:,i,1).*Dphi(:,j,2).*area;
19         k13 = Dphi(:,i,1).*Dphi(:,j,3).*area;
20         k21 = Dphi(:,i,2).*Dphi(:,j,1).*area;
21         k22 = Dphi(:,i,2).*Dphi(:,j,2).*area;
22         k23 = Dphi(:,i,2).*Dphi(:,j,3).*area;
23         k31 = Dphi(:,i,3).*Dphi(:,j,1).*area;
24         k32 = Dphi(:,i,3).*Dphi(:,j,2).*area;

```

```

25         k33 = Dphi(:,i,3).*Dphi(:,j,3).*area;
26         K = [k11,k12,k13,k21,k22,k23,k31,k32,k33]; % stored in rows
27         Dbase{i,j} = K(:); % straighten
28     end
29 end
30
31 % ----- Sparse assembling indices -----
32 nnz = NT*Ndof^2;
33 ii = zeros(nnz,1); jj = zeros(nnz,1);
34 id = 0;
35 for i = 1:Ndof
36     for j = 1:Ndof
37         ii(id+1:id+NT) = elem(:,i); % zi
38         jj(id+1:id+NT) = elem(:,j); % zj
39         id = id + NT;
40     end
41 end
42
43 ii11 = ii; jj11 = jj; ii12 = ii; jj12 = jj+N;
44 ii21 = ii+N; jj21 = jj; ii22 = ii+N; jj22 = jj+N;
45
46 % ----- Assemble stiffness matrix -----
47 ss11 = (lambda+2*mu)*Dbase{1,1} + mu*Dbase{2,2};
48 ss12 = lambda*Dbase{1,2} + mu*Dbase{2,1};
49 ss21 = lambda*Dbase{2,1} + mu*Dbase{1,2};
50 ss22 = (lambda+2*mu)*Dbase{2,2} + mu*Dbase{1,1};
51 ii = [ii11; ii12; ii21; ii22];
52 jj = [jj11; jj12; jj21; jj22];
53 ss = [ss11; ss12; ss21; ss22];
54 kk = sparse(ii,jj,ss,2*N,2*N);
55
56 % ----- Assemble load vector -----
57 % % mid-point quadrature rule
58 % x1 = node(elem(:,1),1); y1 = node(elem(:,1),2);
59 % x2 = node(elem(:,2),1); y2 = node(elem(:,2),2);
60 % x3 = node(elem(:,3),1); y3 = node(elem(:,3),2);
61 % xc = 1/3*(x1+x2+x3); yc = 1/3*(y1+y2+y3); pc = [xc,yc];
62 %
63 % f1 = f(pc).*area./3; f2 = f1; f3 = f1;
64 % F1 = [f1(:,1),f2(:,1),f3(:,1)];

```

```

65 % F2 = [f1(:,2),f2(:,2),f3(:,2)];
66
67 % Gauss quadrature rule
68 [lambda,weight] = quadpts(2);
69 f1 = zeros(NT,2); f2 = f1; f3 = f1;
70 for iel = 1:NT
71     vK = node(elem(iel,:)); % vertices of K
72     xy = lambda*vK; fxy = f(xy); % fxy = [f1xy,f2xy]
73     fv1 = fxy.*[lambda(:,1),lambda(:,1)]; % (f,phi1)
74     fv2 = fxy.*[lambda(:,2),lambda(:,2)]; % (f,phi2)
75     fv3 = fxy.*[lambda(:,3),lambda(:,3)]; % (f,phi3)
76
77     f1(iel,:) = area(iel)*weight*fv1;
78     f2(iel,:) = area(iel)*weight*fv2;
79     f3(iel,:) = area(iel)*weight*fv3;
80 end
81 F1 = [f1(:,1),f2(:,1),f3(:,1)];
82 F2 = [f1(:,2),f2(:,2),f3(:,2)];
83 ff = accumarray([elem(:); elem(:)+N], [F1(:);F2(:)], [2*N 1]);
84
85 % ----- Neumann boundary condition -----
86 elemN = bdStruct.elemN;
87 if ~isempty(elemN)
88     g_N = pde.g_N;
89     z1 = node(elemN(:,1)); z2 = node(elemN(:,2));
90     e = z1-z2; % e = z2-z1
91     ne = [-e(:,2),e(:,1)]; % scaled ne
92     Sig1 = g_N(z1); Sig2 = g_N(z2);
93     F11 = sum(ne.*Sig1(:,[1,3]),2)./2; F12 = ...
94         sum(ne.*Sig2(:,[1,3]),2)./2;
95     F21 = sum(ne.*Sig1(:,[2,3]),2)./2; F22 = ...
96         sum(ne.*Sig2(:,[2,3]),2)./2;
97     FN = [F11,F12,F21,F22];
98     ff = ff + accumarray([elemN(:); elemN(:)+N], FN(:), [2*N 1]);
99 end
100
101 % ----- Dirichlet boundary condition -----
102 g_D = pde.g_D; eD = bdStruct.eD;
103 isBdNode = false(N,1); isBdNode(eD) = true;
104 bdNode = find(isBdNode); freeNode = find(~isBdNode);

```

```

103 pD = node(bdNode,:);
104 bdDof = [bdNode; bdNode+N]; freeDof = [freeNode; freeNode+N];
105 u = zeros(2*N,1); uD = g_D(pD); u(bdDof) = uD(:);
106 ff = ff - kk*u;
107
108 % ----- Solver -----
109 u(freeDof) = kk(freeDof,freeDof)\ff(freeDof);

```

注 6.2 我们也给出了向量有限元空间的程序, 主程序为 main_elasticityVector.m, 这里不再列出.

6.4 第二种形式的变分问题

单元变分形式为

$$\begin{aligned} a_K(\mathbf{u}, \mathbf{v}) &= 2\mu \int_K \varepsilon_{ij}(\mathbf{u}) \varepsilon_{ij}(\mathbf{v}) dx + \lambda \int_K (\partial_i u_i)(\partial_j v_j) dx \\ &=: 2\mu I + \lambda J, \end{aligned}$$

注意求和约定.

第一部分可利用向量法分析, 如下装配

```

1 % ----- Stiffness matrix for (Eij(u):Eij(v)) -----
2 ss11 = Dbase{1,1}+0.5*Dbase{2,2}; ss12 = 0.5*Dbase{2,1};
3 ss21 = 0.5*Dbase{1,2}; ss22 = Dbase{2,2}+0.5*Dbase{1,1};
4 A11 = sparse(ii,jj,ss11,N,N); A12 = sparse(ii,jj,ss12,N,N);
5 A21 = sparse(ii,jj,ss21,N,N); A22 = sparse(ii,jj,ss22,N,N);
6 A = [A11,A12; A21,A22];
7 A = 2*mu*A;

```

根据装配的说明, 上面的分块写法也可直接用 `sparse` 实现, 如下

```

1 ii11 = ii; jj11 = jj; ii12 = ii; jj12 = jj+N;
2 ii21 = ii+N; jj21 = jj; ii22 = ii+N; jj22 = jj+N;
3
4 ss11 = Dbase{1,1}+0.5*Dbase{2,2}; ss12 = 0.5*Dbase{2,1};
5 ss21 = 0.5*Dbase{1,2}; ss22 = Dbase{2,2}+0.5*Dbase{1,1};
6 ii = [ii11; ii12; ii21; ii22];
7 jj = [jj11; jj12; jj21; jj22];
8 ss = [ss11; ss12; ss21; ss22];
9 A = sparse(ii,jj,ss,2*N,2*N);
10 A = 2*mu*A;

```

现在考虑第二部分, 即

$$\lambda J = \lambda \int_K (\partial_i u_i)(\partial_j v_j) dx,$$

注意指标的求和约定. 明确写出来为

$$J = \int_K (\partial_1 v_1 \partial_1 u_1 + \partial_2 v_2 \partial_1 u_1 + \partial_1 v_1 \partial_2 u_2 + \partial_2 v_2 \partial_2 u_2) dx.$$

它的装配可根据前面的分量配对给出.

```
1 % ----- Stiffness matrix for (div u,div v) -----
2 ss11 = Dbase{1,1};           ss12 = Dbase{1,2};
3 ss21 = Dbase{2,1};           ss22 = Dbase{2,2};
4 B11 = sparse(ii,jj,ss11,N,N); B12 = sparse(ii,jj,ss12,N,N);
5 B21 = sparse(ii,jj,ss21,N,N); B22 = sparse(ii,jj,ss22,N,N);
6 B = [B11,B12; B21,B22];
7 B = lambda*B;
```

或直接装配

```
1 % (div u,div v)
2 ss11 = Dbase{1,1};           ss12 = Dbase{1,2};
3 ss21 = Dbase{2,1};           ss22 = Dbase{2,2};
4 ss = [ss11; ss12; ss21; ss22];
5 B = sparse(ii,jj,ss,2*N,2*N);
6 B = lambda*B;
```

其他过程与第一种形式相同, 不再列出 (主程序为 main_elasticity2.m).

6.5 第三种形式的变分问题

变分形式 (6.8) 的程序只需要对前面的进行简单修改即可. 需要注意的是, 此种形式产生的边界项为

$$\mu \int_{\partial\Omega} \partial_n \mathbf{u} \cdot \mathbf{v} ds + (\lambda + \mu) \int_{\partial\Omega} (\operatorname{div} \mathbf{u})(\mathbf{v} \cdot \mathbf{n}) ds,$$

给出相应的边界项我们也可容易地处理边界项. 但弹性问题中通常不附加上面类型的边界条件, 为此, 第三种形式只考虑 Dirichlet 边界条件.

主程序为 (数据文件为 elasticitydata1.m, L 型区域, 用 MATLAB 的 PDE 工具箱生成)

CODE 6.4. main_elasticityLaplace.m

```
1 clc;clear;close all
2 % ----- Mesh and boudary conditions -----
3 g = [2 2 2 2 2 2    % decomposed geometry matrix
```

```

4      0   1   1  -1  -1   0
5      1   1  -1  -1   0   0
6      0   0   1   1  -1  -1
7      0   1   1  -1  -1   0
8      1   1   1   1   1   1
9      0   0   0   0   0   0];
10 [p,e,t] = initmesh(g, 'hmax',1); % initial mesh
11 for i = 1:3
12     [p,e,t] = refinemesh(g,p,e,t); % refine mesh
13 end
14 node = p'; elem = t(1:3,:)';
15 figure, showmesh(node,elem);
16 bdStruct = setboundary(node,elem);
17
18 % ----- PDE data -----
19 lambda = 1; mu = 1;
20 para.lambda = lambda; para.mu = mu;
21 pde = elasticitydata1(para);
22
23 % ----- elasticity -----
24 u = elasticityLaplace(node,elem,pde,bdStruct);
25
26 % ----- error analysis -----
27 u = reshape(u,[],2);
28 uexact = pde.uexact; ue = uexact(node);
29 id = 1;
30 figure,
31 subplot(1,2,1), showsolution(node,elem,u(:,id));
32 zlabel('u');
33 subplot(1,2,2), showsolution(node,elem,ue(:,id));
34 zlabel('ue');
35 Eabs = u-ue; % Absolute errors
36 figure, showsolution(node,elem,Eabs(:,id)); zlim('auto');
37 format shorte
38 Err = norm(Eabs)./norm(ue)

```

结果为

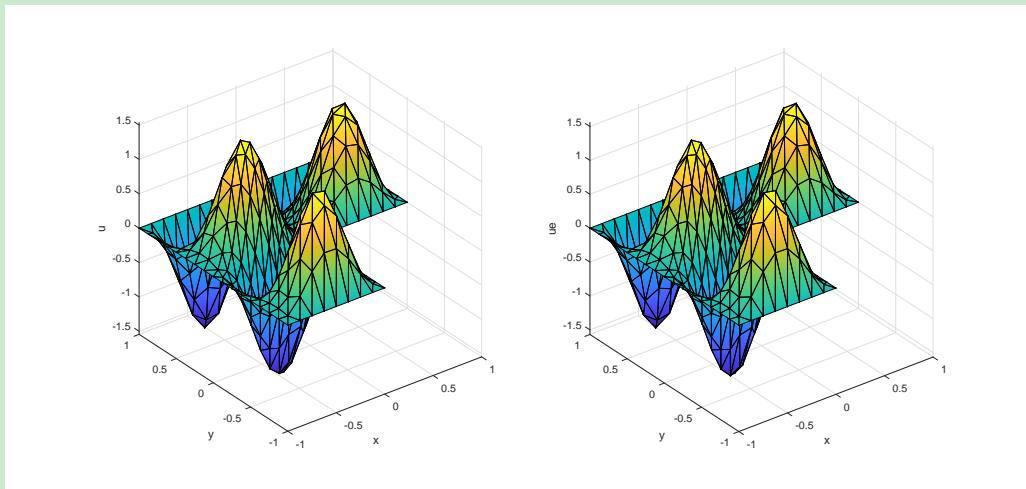
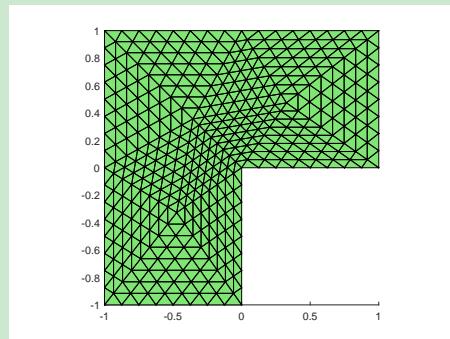


图 6.2. 变分形式 (6.8) 的数值解与精确解

相应的网格剖分如下



变分形式 (6.8) 的程序只需要对前面的进行简单修改即可, 如下

CODE 6.5. elasticityLaplace.m

```

1 function u = elasticityLaplace(node, elem, pde, bdFlag)
2 %Elasticity Conforming P1 elements discretization of linear ...
3 % elasticity equation
4 %
5 %      u = [u1, u2]
6 %      -mu \Delta u - (lambda + mu)*grad(div(u)) = f in \Omega
7 %      Dirichlet boundary condition u = [g1_D, g2_D] on \Gamma_D.
8 %
9 N = size(node,1); NT = size(elem,1); Ndof = 3;
10 mu = pde.mu; lambda = pde.lambda; f = pde.f;
11 % ----- Compute (Dbase,Djbase) -----
12 [Dphi,area] = gradbasis(node,elem);
13 Dbase = cell(2,2);

```

```

14 for i = 1:2
15     for j = 1:2
16         k11 = Dphi(:,i,1).*Dphi(:,j,1).*area;
17         k12 = Dphi(:,i,1).*Dphi(:,j,2).*area;
18         k13 = Dphi(:,i,1).*Dphi(:,j,3).*area;
19         k21 = Dphi(:,i,2).*Dphi(:,j,1).*area;
20         k22 = Dphi(:,i,2).*Dphi(:,j,2).*area;
21         k23 = Dphi(:,i,2).*Dphi(:,j,3).*area;
22         k31 = Dphi(:,i,3).*Dphi(:,j,1).*area;
23         k32 = Dphi(:,i,3).*Dphi(:,j,2).*area;
24         k33 = Dphi(:,i,3).*Dphi(:,j,3).*area;
25         K = [k11,k12,k13,k21,k22,k23,k31,k32,k33]; % stored in rows
26         Dbase{i,j} = K(:); % straighten
27     end
28 end
29
30 % ----- Sparse assembling indices -----
31 nnz = NT*Ndof^2;
32 ii = zeros(nnz,1); jj = zeros(nnz,1);
33 id = 0;
34 for i = 1:Ndof
35     for j = 1:Ndof
36         ii(id+1:id+NT) = elem(:,i); % zi
37         jj(id+1:id+NT) = elem(:,j); % zj
38         id = id + NT;
39     end
40 end
41
42 ii11 = ii; jj11 = jj; ii12 = ii; jj12 = jj+N;
43 ii21 = ii+N; jj21 = jj; ii22 = ii+N; jj22 = jj+N;
44
45 % ----- Assemble stiffness matrix -----
46 % (grad u,grad v)
47 ss11 = Dbase{1,1}+Dbase{2,2}; ss22 = ss11;
48 % A11 = sparse(ii,jj,ss11,N,N); A12 = sparse(N,N);
49 % A21 = sparse(N,N); A22 = sparse(ii,jj,ss22,N,N);
50 % A = [A11,A12; A21,A22];
51 ii = [ii11; ii22]; jj = [jj11; jj22]; ss = [ss11; ss22];
52 A = sparse(ii,jj,ss,2*N,2*N);
53 A = mu*A;

```

```

54
55 % (div u,div v)
56 ss11 = Dbase{1,1}; ss12 = Dbase{1,2};
57 ss21 = Dbase{2,1}; ss22 = Dbase{2,2};
58 % B11 = sparse(ii,jj,ss11,N,N); B12 = sparse(ii,jj,ss12,N,N);
59 % B21 = sparse(ii,jj,ss21,N,N); B22 = sparse(ii,jj,ss22,N,N);
60 % B = [B11,B12; B21,B22];
61 ii = [ii11; ii12; ii21; ii22];
62 jj = [jj11; jj12; jj21; jj22];
63 ss = [ss11; ss12; ss21; ss22];
64 B = sparse(ii,jj,ss,2*N,2*N);
65 B = (lambda+mu)*B;
66
67 % stiffness matrix
68 kk = A + B;
69
70 % ----- Assemble load vector -----
71 % Gauss quadrature rule
72 [lambda,weight] = quadpts(2);
73 f1 = zeros(NT,2); f2 = f1; f3 = f1;
74 for iel = 1:NT
75     vK = node(elem(iel,:)); % vertices of K
76     xy = lambda*vK; fxy = f(xy); % fxy = [f1xy,f2xy]
77     fv1 = fxy.*[lambda(:,1),lambda(:,1)]; % (f,phi1)
78     fv2 = fxy.*[lambda(:,2),lambda(:,2)]; % (f,phi2)
79     fv3 = fxy.*[lambda(:,3),lambda(:,3)]; % (f,phi3)
80
81     f1(iel,:) = area(iel)*weight*f1;
82     f2(iel,:) = area(iel)*weight*f2;
83     f3(iel,:) = area(iel)*weight*f3;
84 end
85 F1 = [f1(:,1),f2(:,1),f3(:,1)];
86 F2 = [f1(:,2),f2(:,2),f3(:,2)];
87 % ff1 = accumarray(elem(:), F1(:), [N 1]);
88 % ff2 = accumarray(elem(:), F2(:), [N 1]);
89 % ff = [ff1;ff2];
90 ff = accumarray([elem(:); elem(:)+N], [F1(:);F2(:)], [2*N 1]);
91
92 % ----- Dirichlet boundary condition -----
93 g_D = pde.g_D; eD = bdFlag.eD;

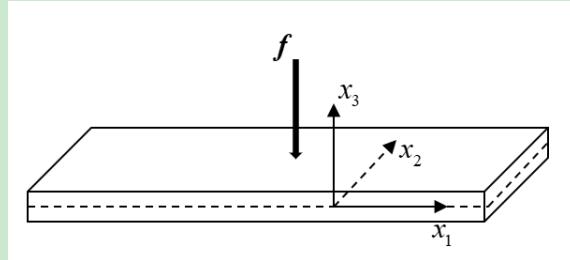
```

```
94 isBdNode = false(N,1); isBdNode(eD) = true;
95 bdNode = find(isBdNode); freeNode = find(~isBdNode);
96 pD = node(bdNode,:);
97 bdDof = [bdNode; bdNode+N]; freeDof = [freeNode; freeNode+N];
98 u = zeros(2*N,1); uD = g_D(pD); u(bdDof) = uD(:);
99 ff = ff - kk*u;
100
101 % ----- Solver -----
102 u(freeDof) = kk(freeDof,freeDof)\ff(freeDof);
```

第七章 Kirchhoff 板弯问题

7.1 平衡方程、变分问题与边界条件

7.1.1 平衡方程与边界条件



给定一个薄板, 即厚度 t 远小于水平方向的尺度. 称板厚方向为横向, 水平方向为纵向. 在板厚方向施加横向的载荷, 薄板会发生弯曲 (小变形), 且设板厚中面的位移, 即挠度为 w .

在弹性力学基本假设以及 Kirchhoff 计算假设下, 薄板弯曲问题可简化为挠度 w 的平面问题, 平衡方程为

$$\Omega : \quad -\partial_{ij} M_{ij}(w) = f, \quad (7.1)$$

式中,

$$M_{ij} = D ((1 - \nu) K_{ij} + \nu K_{kk} \delta_{ij}), \quad K_{ij} = -\partial_{ij} w,$$

指标范围为 $i, j = 1, 2, k \in \{1, 2\}$. 该方程是挠度 w 的四阶椭圆型偏微分方程. 若板面与地基有弹性耦合, 则平衡方程可改写为

$$\Omega : \quad -\partial_{ij} M_{ij}(w) + cw = f,$$

其中 c 为弹性耦合常数.

当 D, ν 为常数时, 含有 ν 的项会抵消, 上述方程简化为双调和方程

$$D\Delta^2 w = f.$$

注意, 对任意的 ν 都是如此, 特别地, 取 $\nu = 0$ (工程中是不允许的), 原来的平衡方程就是 $D\partial_{ij}\partial_{ij}w = f$, 显然 $\partial_{ij}\partial_{ij} = \Delta^2$. 为了方便, 以下将坐标 (x_1, x_2) 改记为 (x, y) .

本文考虑强加边界条件

$$w = \partial_n w = 0, \quad (x, y) \in \partial\Omega.$$

7.1.2 变分问题

在平衡方程 (7.1) 两边乘以检验函数 v 并分部积分后可获得变分问题, 此时会出现复杂的边界项. 但当 $v \in H_0^2(\Omega)$ 时, 这些边界项都会消失.

变分问题为: 找 $u \in V := H_0^2(\Omega)$ 使得

$$D(w, v) = F(v), \quad v \in V,$$

式中,

$$\begin{aligned} D(w, v) &= \int_{\Omega} M_{ij}(w) K_{ij}(v) dx dy + \int_{\Omega} c w v dx dy, \\ F(v) &= \int_{\Omega} f v dx dy. \end{aligned}$$

命

$$M = \begin{bmatrix} M_{11} \\ M_{22} \\ M_{12} \end{bmatrix}, \quad K = \begin{bmatrix} K_{11} \\ K_{22} \\ 2K_{12} \end{bmatrix},$$

则

$$\int_{\Omega} M_{ij}(w) K_{ij}(v) dx dy = \int_{\Omega} K^T(v) M(w) dx dy,$$

且薄板弯曲的 Hooke 定律可写为

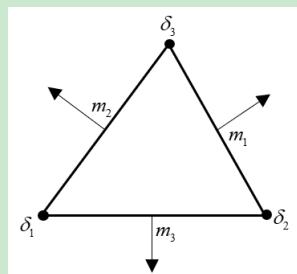
$$M = R K, \quad R = D \begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & (1-\nu)/2 \end{bmatrix}. \quad (7.2)$$

7.2 非协调 Morley 元

7.2.1 Morley 元的构造

Morley 元也称为完全二次三角形元.

局部节点基



Morley 元的三元组 $(K, \mathcal{P}, \mathcal{N})$ 中的

$$K = \beta \text{ 为三角形,}$$

$$\mathcal{P} = \mathbb{P}_2(K),$$

$$\mathcal{N} = \{v(p), p = \delta_1, \delta_2, \delta_3; \partial_n v(m), m = m_1, m_2, m_3\}.$$

注 7.1 注意 $\partial_n v = \nabla v \cdot \vec{n}$, 对相邻两个单元来说, $\partial_n v(m_i)$ 是不同的. 事实上, 它跨边界仍是连续的, 从而

$$\partial_n v(m_i)|_{\beta_1} = -\partial_n v(m_i)|_{\beta_2},$$

其中, m_i 是相邻三角形 β_1 和 β_2 公共边的中点. 对该点来说, 整体自由度只要取定一个方向即可, 注意基函数的问题会相差符号, 后面说明.

注 7.2 边中点法向导数也可换为如下的积分平均

$$\frac{1}{|e_i|} \int_{e_i} \partial_n v \, ds,$$

此时基函数和有限元空间不变 (仍有方向性问题).

现在考虑局部节点基. 取三角形 $\beta = (\delta_1, \delta_2, \delta_3)$, 三对边的中点分别记为 m_1, m_2, m_3 . 以这 6 个点作为插值点, 构造一个完全二次多项式

$$Q(x, y) = a_1 + a_2x + a_3y + a_4x^2 + a_5xy + a_6y^2,$$

要求在顶点上的函数值以及三边中点上法向导数值相同, 即

$$Q(\delta_i) = w_i, \quad i = 1, 2, 3; \quad Q_n(m_i) = w_{mi}, \quad i = 1, 2, 3,$$

这 6 个条件可唯一确定多项式的 6 个系数. 设节点基函数分别为 $\varphi_i, \psi_i(x, y)$, $i = 1, 2, 3$, 即

$$Q(x, y) = \sum_{i=1}^3 w_i \varphi_i(x, y) + \sum_{i=1}^3 w_{mi} \psi_i(x, y), \quad (7.3)$$

满足

- 对 $i = 1, 2, 3$, 有

$$\varphi_i(\delta_j) = \delta_{ij}, \quad j = 1, 2, 3; \quad \partial_n \varphi_i(m_j) = 0, \quad j = 1, 2, 3.$$

- 对 $i = 4, 5, 6$, 有

$$\varphi_i(\delta_j) = 0, \quad j = 1, 2, 3; \quad \partial_n \varphi_i(m_j) = \delta_{ij}, \quad j = 1, 2, 3.$$

由这些条件, 可用重心坐标来构造基函数 φ_i . 设重心坐标为 $\lambda_1, \lambda_2, \lambda_3$, i, j, k 表示 1,2,3 的轮换. 定义

$$\xi_i = x_j - x_k, \quad \eta_i = y_j - y_k, \quad \omega_i = x_j y_k - x_k y_j,$$

则三角形的有向面积可表示为

$$S = \frac{1}{2} \det \begin{bmatrix} 1 & x_1 & y_1 \\ 1 & x_2 & y_2 \\ 1 & x_3 & y_3 \end{bmatrix} = \frac{1}{2} (\xi_1 \eta_2 - \xi_2 \eta_1) = \omega_1 \omega_2 \omega_3,$$

且

$$\lambda_i(x, y) = \frac{1}{2S} (\eta_i x - \xi_i y + \omega_i), \quad i = 1, 2, 3. \quad (7.4)$$

从 (x, y) 到 (λ_1, λ_2) 变换的 Jacobi 矩阵的行列式为

$$|J| = \det \frac{\partial(\lambda_1, \lambda_2)}{\partial(x, y)} = \frac{1}{2S}.$$

再令

$$s_{ij} = -(\xi_i \xi_j + \eta_i \eta_j),$$

则有

$$\left\{ \begin{array}{l} \varphi_1 = \lambda_1^2 + \left(\frac{s_{12}}{l_2^2} + \frac{s_{31}}{l_3^2} \right) \lambda_2 \lambda_3 + \frac{s_{31}}{l_3^2} \lambda_3 \lambda_1 + \frac{s_{12}}{l_2^2} \lambda_1 \lambda_2, \\ \varphi_2 = \lambda_2^2 + \frac{s_{23}}{l_3^2} \lambda_2 \lambda_3 + \left(\frac{s_{23}}{l_3^2} + \frac{s_{12}}{l_1^2} \right) \lambda_3 \lambda_1 + \frac{s_{12}}{l_1^2} \lambda_1 \lambda_2, \\ \varphi_3 = \lambda_3^2 + \frac{s_{23}}{l_2^2} \lambda_2 \lambda_3 + \frac{s_{31}}{l_1^2} \lambda_3 \lambda_1 + \left(\frac{s_{31}}{l_1^2} + \frac{s_{23}}{l_2^2} \right) \lambda_1 \lambda_2, \\ \psi_1 = \frac{2S}{l_1} \lambda_1 (\lambda_1 - 1), \quad \psi_2 = \frac{2S}{l_2} \lambda_2 (\lambda_2 - 1), \quad \psi_3 = \frac{2S}{l_3} \lambda_3 (\lambda_3 - 1), \end{array} \right.$$

式中, l_i 表示第 i 个顶点所对边的边长 (注意基函数中指标的轮换).

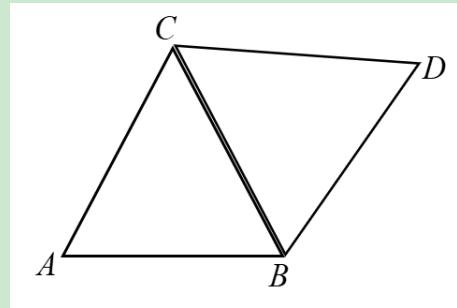
整体节点基

前面已经指出了边中点自由度的方向性问题. 不考虑边界条件, Morley 元的整体有限元空间为

$$V_h = \{v \in L^2(\Omega) : v \text{ 的 Morley 元自由度连续}\}.$$

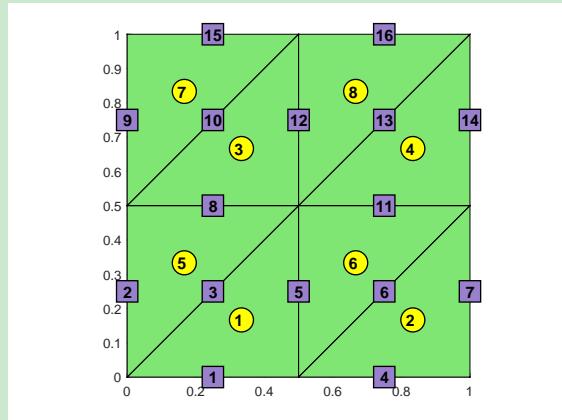
对内部边 e , 可任意取定一个三角形的外法向量参数作为整体节点参数, 不妨称其为定参三角形. 此时, 整体节点基限制在定参三角形上就是局部节点基, 而限制在相邻三角形上则与局部节点基相差负号.

从基函数中可以看到, ψ_i 中的 l_i 可起到变号的作用, 即可规定 l_i 是有向长度 (此时 φ 不会受到影响).



1. 边的定向: 对三角形单元的边 e , 规定逆时针方向为其在单元中的正向, 可用局部编号来确定. 例如, 设 ΔABC 的第 1 个顶点为 A , 则 BC 的局部编号为 2-3.
2. 边的符号差:
 - 对内部定向边, 称终点与起点整体编号差的符号为该边在单元中的符号差.
 - 对边界边, 规定符号差为正 (即 1).
 - 显然对内部边, 限制在相邻两个单元上的符号差异号.

下面说明如何获得边的符号差, 用矩阵 `signelem` 存储, 它是按单元给出的.



- 辅助数据结构中给出了 `elem2edge`, 它按单元存储每条边的自然序号 (符合第 i 个顶点对边为第 i 条边), 而 `edge` 存储了一维边的端点编号. 可如下计算按单元存储的边的长度

```

1 % edge length
2 z1 = node(edge(:,1),:); z2 = node(edge(:,2),:);
3 he = sqrt(sum((z2-z1).^2,2)); L = he(elem2edge);

```

- 现在给边添上符号差. 执行如下语句可获得内部边的符号差

```

1 signelem = sign([elem(:,3)-elem(:,2), elem(:,1)-elem(:,3), ...
    elem(:,2)-elem(:,1)]);

```

这里边界边的符号差可能不正确.

- 利用逻辑运算可修改 `sgnL` 中边界边的符号差, 从而获得带符号的边长

```
1 bdIndex = bdStruct.bdIndex;
2 E = false(NE,1); E(bdIndex) = 1; sgnbd = E(elem2edge);
3 sgnelem(sgnbd) = 1;
4 sgnL = sgnelem.*L;
```

这里, `bdIndex` 是边界边的自然序号, `E` 内部边对应 `false`, 边界边对应 `true`.

7.2.2 sparse 装配指标

为了方便, 以下将基函数按顺序编号为 $\varphi_1, \dots, \varphi_6$, 这里,

$$\varphi_{3+i} = \psi_i, \quad i = 1, 2, 3.$$

单元变量为 $W_\beta = [w_1, \dots, w_6]^T$, 前三个对应三角形顶点, 后三个对应边的中点, 为此要对边的中点进行编号. 在辅助数据结构中, 我们给出了两个与边相关的数据结构, 它们分别是

- `edge`: 一维边的端点标记;
- `elem2edge`: 边的自然序号 (按单元存储).

将一维边接着顶点进行编号, 根据单元自由度的排序规则, 显然新的连通性信息为

```
1 elem2 = [elem, elem2edge+N];
```

类似 Poisson 方程, `sparse` 装配指标为

```
1 % ----- Sparse assembling indices -----
2 Ndof = 6;
3 nnz = NT*Ndof^2;
4 ii = zeros(nnz,1); jj = zeros(nnz,1);
5 id = 0;
6 for i = 1:Ndof
7     for j = 1:Ndof
8         ii(id+1:id+NT) = elem2(:,i);
9         jj(id+1:id+NT) = elem2(:,j);
10        id = id + NT;
11    end
12 end
```

设单元刚度矩阵如下排列,

```
1 K = [k11, k12, ..., k66];
```

则刚度矩阵的装配与 Poisson 方程类似, 即

```
1 kk = sparse(ii, jj, K(:, N+NE, N+NE));
```

注意变量个数为 $N+NE$ 个, 其中 NE 是一维边的个数.

对载荷向量, 设其如下排列

```
1 F = [F1, F2, F3, F4, F5, F6];
```

则载荷向量如下装配

```
1 ff = accumarray(elem2(:, F(:, [N+NE 1]));
```

7.2.3 刚度矩阵与载荷向量的计算

双线性形式第一项的计算

记

$$W_\beta = [w_1, \dots, w_6]^T, \quad N = [\varphi_1, \dots, \varphi_6],$$

注意基函数采用行向量写法 (回忆线性代数), 则 (7.3) 中的插值函数 Q 可写为

$$Q = NW_\beta.$$

由 (7.2), 只需要给出 K 的近似. 显然有

$$K = \begin{bmatrix} K_{11} \\ K_{22} \\ 2K_{12} \end{bmatrix} = - \begin{bmatrix} \partial_{11}w \\ \partial_{22}w \\ 2\partial_{12}w \end{bmatrix} \approx - \begin{bmatrix} \partial_{11}Q \\ \partial_{22}Q \\ 2\partial_{12}Q \end{bmatrix} = - \begin{bmatrix} \partial_{11}N \\ \partial_{22}N \\ 2\partial_{12}N \end{bmatrix} W_\beta =: BW_\beta,$$

其中,

$$B = - \begin{bmatrix} \partial_{11}N \\ \partial_{22}N \\ 2\partial_{12}N \end{bmatrix} = [B_1, \dots, B_6], \quad B_i = - \begin{bmatrix} \partial_{11}\varphi_i \\ \partial_{22}\varphi_i \\ 2\partial_{12}\varphi_i \end{bmatrix}.$$

双线性形式的第一项为

$$\begin{aligned} & \int_{\beta} M_{ij}(w) K_{ij}(v) dx dy \\ &= \int_{\beta} K^T(v) M(w) dx dy = \int_{\beta} K^T(v) R K(w) dx dy \\ &= V_\beta^T \int_{\beta} B^T R B dx dy W_\beta =: V_\beta^T K_\beta W_\beta, \end{aligned}$$

式中,

$$K_\beta = \int_{\beta} B^T R B dx dy = (K_{ij})_{6 \times 6}, \quad K_{ij} = \int_{\beta} B_i^T R B_j dx dy.$$

显然 B 是常数矩阵, 且有

$$K_{ij} = \int_{\beta} B_i^T R B_j dx dy = |\beta| B_i^T R B_j = |\beta| D \cdot (k_{ij})_{6 \times 6},$$

其中 k_{ij} 是常数, 满足

$$k_{ij} = \partial_{11}\varphi_i \partial_{11}\varphi_j + \partial_{22}\varphi_i \partial_{22}\varphi_j + \nu(\partial_{11}\varphi_i \partial_{22}\varphi_j + \partial_{22}\varphi_i \partial_{11}\varphi_j) + 2(1-\nu)\partial_{12}\varphi_i \partial_{12}\varphi_j.$$

直接计算有

$$\begin{cases} \partial_{11}\varphi_1 = \frac{1}{2S^2} \left(\eta_1^2 - \frac{s_{12}}{l_2^2} \eta_2^2 - \frac{s_{31}}{l_3^2} \eta_3^2 \right), \\ \partial_{22}\varphi_1 = \frac{1}{2S^2} \left(\xi_1^2 - \frac{s_{12}}{l_2^2} \xi_2^2 - \frac{s_{31}}{l_3^2} \xi_3^2 \right), \\ \partial_{12}\varphi_1 = -\frac{1}{2S^2} \left(\xi_1 \eta_1 - \frac{s_{12}}{l_2^2} \xi_2 \eta_2 - \frac{s_{31}}{l_3^2} \xi_3 \eta_3 \right), \end{cases}$$

对 φ_2 的各阶导数, 在上式中把右端的 $(1, 2, 3) \rightarrow (2, 3, 1)$; 对 φ_3 的各阶导数, 在上式中把右端的 $(1, 2, 3) \rightarrow (3, 1, 2)$. 而

$$\begin{cases} \partial_{11}\varphi_{3+i} = \frac{1}{Sl_i} \eta_i^2 \\ \partial_{22}\varphi_{3+i} = \frac{1}{Sl_i} \xi_i^2 \quad , \quad i = 1, 2, 3. \\ \partial_{12}\varphi_{3+i} = -\frac{1}{Sl_i} \xi_i \eta_i \end{cases}$$

1. 计算中用到的 ξ_i, η_i 如下获得

```

1 auxG = auxgeometry(node, elem); area = auxG.area;
2 x1 = node(elem(:, 1), 1); y1 = node(elem(:, 1), 2);
3 x2 = node(elem(:, 2), 1); y2 = node(elem(:, 2), 2);
4 x3 = node(elem(:, 3), 1); y3 = node(elem(:, 3), 2);
5 % xi, eta
6 xi = [x2-x3, x3-x1, x1-x2]; eta = [y2-y3, y3-y1, y1-y2];

```

所有单元的 s_{ij} 用三维数组存储

```

1 % sij
2 sb = zeros(NT, 3, 3);
3 for i = 1:3
4     j = 1:3;
5     sb(:, i, j) = -xi(:, i).*xi(:, j) - eta(:, i).*eta(:, j);
6 end

```

也可用元胞数组存储, 但不利于向量化运算.

2. 现在计算基函数的二阶导数. 我们用 b_{11} 存储所有基函数的 ∂_{11} 导数, 即

$$b_{11} = [\partial_{11}\varphi_1, \partial_{11}\varphi_2, \dots, \partial_{11}\varphi_6],$$

且每行对应一个单元.

```
1 % second derivatives of phi_i
2 b11 = zeros(NT,6); b22 = b11; b12 = b11; % [phi_i, i=1:6]
3 ind = [1 2 3; 2 3 1; 3 1 2]; % cyclic permutation
4 for i = 1:3
5     j = ind(i,2); k = ind(i,3);
6     c0 = 1./(2*area.^2);
7     c1 = sb(:,i,j)./sgnL(:,j).^2;
8     c2 = sb(:,k,i)./sgnL(:,k).^2;
9     % i = 1,2,3
10    b11(:,i) = c0.* (eta(:,i).^2 - c1.*eta(:,j).^2 - ...
11                  c2.*eta(:,k).^2);
12    b22(:,i) = c0.* (xi(:,i).^2 - c1.*xi(:,j).^2 - ...
13                  c2.*xi(:,k).^2);
14    b12(:,i) = -c0.* (xi(:,i).*eta(:,i) - ...
15                      c1.*xi(:,j).*eta(:,j) - c2.*xi(:,k).*eta(:,k));
16    % i = 4,5,6
17    ci = 1./(area.*sgnL(:,i));
18    b11(:,3+i) = ci.*eta(:,i).^2;
19    b22(:,3+i) = ci.*xi(:,i).^2;
20    b12(:,3+i) = -ci.*xi(:,i).*eta(:,i);
21 end
```

注意这里用指标的轮换进行计算. 上面的循环可以避免, 只不过系数 c_0, c_1, c_2 可能要用循环生成, 如下

```
1 % second derivatives of phi_i
2 b11 = zeros(NT,6); b22 = b11; b12 = b11; % [phi_i, i=1:6]
3 ind = [1 2 3; 2 3 1; 3 1 2]; % rotation index
4 c0 = zeros(NT,3); c1 = c0; c2 = c0;
5 for i = 1:3
6     j = ind(i,2); k = ind(i,3);
7     c0(:,i) = 1./(2*area.^2);
8     c1(:,i) = sb(:,i,j)./sgnL(:,j).^2;
```

```

9      c2(:,i) = sb(:,k,i)./sgnL(:,k).^2;
10 end
11 i = ind(:,1); j = ind(:,2); k = ind(:,3);
12 % i = 1,2,3
13 b11(:,i) = c0.* (eta(:,i).^2 - c1.*eta(:,j).^2 - c2.*eta(:,k).^2);
14 b22(:,i) = c0.* (xi(:,i).^2 - c1.*xi(:,j).^2 - c2.*xi(:,k).^2);
15 b12(:,i) = -c0.* (xi(:,i).*eta(:,i) - c1.*xi(:,j).*eta(:,j) - ...
16 % i = 4,5,6
17 ci = 1./(area.*sgnL(:,i));
18 b11(:,3+i) = ci.*eta(:,i).^2;
19 b22(:,3+i) = ci.*xi(:,i).^2;
20 b12(:,3+i) = -ci.*xi(:,i).*eta(:,i);

```

3. 第一项对应的单元刚度矩阵为

```

1 K = zeros(NT,Ndof^2); s = 1;
2 for i = 1:Ndof
3     for j = 1:Ndof
4         K(:,s) = b11(:,i).*b11(:,j) + b22(:,i).*b22(:,j) ...
5             + para.nu*(b11(:,i).*b22(:,j) + ...
6                 b22(:,i).*b11(:,j)) ...
7             + 2*(1-para.nu)*b12(:,i).*b12(:,j);
8     s = s+1;
9 end
10 K = para.D*area.*K;

```

上面的第二个循环可去掉, 如下

```

1 K = zeros(NT,Ndof^2);
2 for i = 1:Ndof
3     j = 1:Ndof; jd = (i-1)*Ndof+1:i*Ndof;
4     K(:,jd) = b11(:,i).*b11(:,j) + b22(:,i).*b22(:,j) ...
5         + para.nu*(b11(:,i).*b22(:,j) + b22(:,i).*b11(:,j)) ...
6         + 2*(1-para.nu)*b12(:,i).*b12(:,j);
7 end
8 K = para.D*area.*K;

```

双线性形式第二项的计算

双线性形式的第二项为

$$\int_{\beta} c w v dxdy = V_{\beta}^T \int_{\beta} c N^T N dxdy W_{\beta} = V_{\beta}^T G_{\beta} W_{\beta},$$

其中地基能量阵

$$G_{\beta} = \int_{\beta} c N^T N dxdy = (g_{ij})_{6 \times 6}, \quad g_{ij} = \int_{\beta} c \varphi_i \varphi_j dxdy.$$

当 c 是常数时, 可用 Euler 公式计算积分

$$\int_{\beta} \lambda_1^{p_1} \lambda_2^{p_2} \lambda_3^{p_3} dxdy = 2S \frac{p_1! p_2! p_3!}{(p_1 + p_2 + p_3 + 2)!}.$$

以下用三角形上的 Gauss 公式计算, 参考前面的说明, 如节 5.2.1.

- (a) 对固定单元序号 iel , 我们用 `base` 存储所有基函数积分节点处的值, 每列对应一个基函数, 如下

```
1 % Gauss quadrature rule
2 [lambda, weight] = quadpts(2);
3 base = zeros(length(weight), Ndof);
4 for i = 1:3
5     j = ind(i, 2); k = ind(i, 3);
6     c1 = sb(iel, i, j)/sgnL(iel, j)^2; c2 = ...
7         sb(iel, k, i)/sgnL(iel, k)^2;
8     c3 = c1+c2;      ci = 2*area(iel)/sgnL(iel, i);
9     % i = 1, 2, 3
10    base(:, i) = lambda(:, i).^2 + c3*lambda(:, j).*lambda(:, k) ...
11        + c2*lambda(:, k).*lambda(:, i) + ...
12            c1*lambda(:, i).*lambda(:, j);
13
14    % i = 4, 5, 6
15    base(:, 3+i) = ci*lambda(:, i).*(lambda(:, i)-1);
16
17 end
```

类似第一部分, 上面的循环也可去掉.

- (b) 刚度矩阵 G 如下计算

```
1 for i = 1:Ndof
2     j = 1:Ndof; jd = (i-1)*Ndof+1:i*Ndof;
3     gs = cxy.*base(:, i).*base(:, j);
```

```

4      G(iel, jd) = area(iel)*weight*gs;
5 end
```

注意 weight 是行向量. 这里, cxy 是系数 c 在积分节点处的值. 若它是常数, 则如下转化为匿名函数

```

1 if isa(para.c, 'double'), cf = @(xy) para.c+0*xy(:,1); end
```

(c) 循环即可获得 G , 最终如下装配

```

1 kk = sparse(ii, jj, K(:)+G(:), N+NE, N+NE);
```

注 7.3 当 $c = 0$ 时, 上面的一些计算可以去掉, 但其过程在载荷向量的计算中用到.

载荷向量的计算

右端

$$\int_{\beta} f v dx dy = V_{\beta}^T \int_{\beta} f N^T dx dy = V_{\beta}^T F_{\beta},$$

其中,

$$F_{\beta} = \int_{\beta} f N dx dy = (F_i)_{6 \times 1}, \quad F_i = \int_{\beta} f \varphi_i dx dy.$$

载荷向量可如下计算

```

1 for i = 1:Ndof
2     fvi = fxy.*base(:,i);
3     F(iel,i) = area(iel)*weight*fvi;
4 end
```

或用向量化运算

```

1 fv = fxy.*base;
2 F(iel,:) = area(iel)*weight*fv;
```

装配前面已经说明.

7.2.4 边界条件的处理

我们考虑的是强加边界条件

$$w = \partial_n w = 0, \quad (x, y) \in \partial\Omega,$$

这里可以是非齐次的 (这两个条件都是 Dirichlet 边界条件).

$w|_{\partial\Omega} = g$ 与常规情形一样处理. 现考虑 $\partial_n w|_{\partial\Omega} = g_n$.

1. Morley 元构造时, 三角形边上中点处的法向导数值是自由度. 这样, 第二个条件可直接替换已知法向导数值, 这与第一个条件是类似的.
2. 在 setboundary.m 中我们给出了 Dirichlet 点的编号 e_D 以及边界边的编号 $bdIndex$, 这样约束点的编号为

```

1 eD = bdStruct.eD; elemD = bdStruct.elemD;
2 bdIndex = bdStruct.bdIndex; % indices of boundary edges
3 id = [eD; bdIndex+N];

```

由此给出自由点编号

```

1 isBdDof = false(N+NE,1); isBdDof(id) = true;
2 bdDof = find(isBdDof); freeDof = find(~isBdDof);

```

3. 边界节点值为

```

1 g_D = pde.g_D;
2 pD = node(eD,:); wD = g_D(pD);

```

边界边中点法向导数值如下计算

```

1 Dw = pde.Dw;
2 z1 = node(elemD(:,1),:); z2 = node(elemD(:,2),:); zc = (z1+z2)./2;
3 e = z1-z2; % e = z2-z1
4 ne = [-e(:,2),e(:,1)]; ne = ne./he(bdIndex);
5 wnD = sum(Dw(zc).*ne,2);

```

这里 Dw 是梯度 ∇w , 而 $\partial_n w = \nabla w \cdot \vec{n}$ 中的 \vec{n} 通过边界边的旋转获得 (并进行单位化).

4. 这样, 我们如下求解

```

1 w = zeros(N+NE,1); w(bdDof) = [wD;wnD];
2 ff = ff - kk*w;
3 w(freeDof) = kk(freeDof,freeDof)\ff(freeDof);
4 w = w(1:N);

```

7.2.5 数值结果

主程序为 main_PlateBendingMorley.m, 相应的函数文件为 PlateBendingMorley.m, 用到的数据文件为 PlateBendingData.m. 这里不再给出具体程序, 参见 GitHub 上传程序 (elasticity 文件夹中).

区域 $[0, 1]^2$ 的结果如下

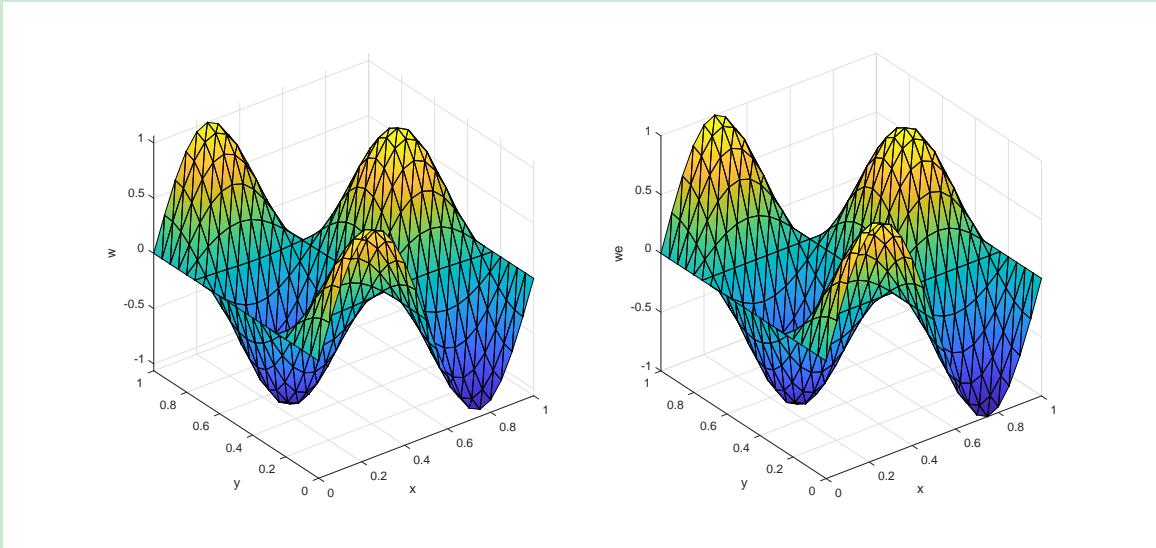


图 7.1. Morley 元方法的数值解与精确解 ($N_x=N_y=20$)

绝对误差见下图

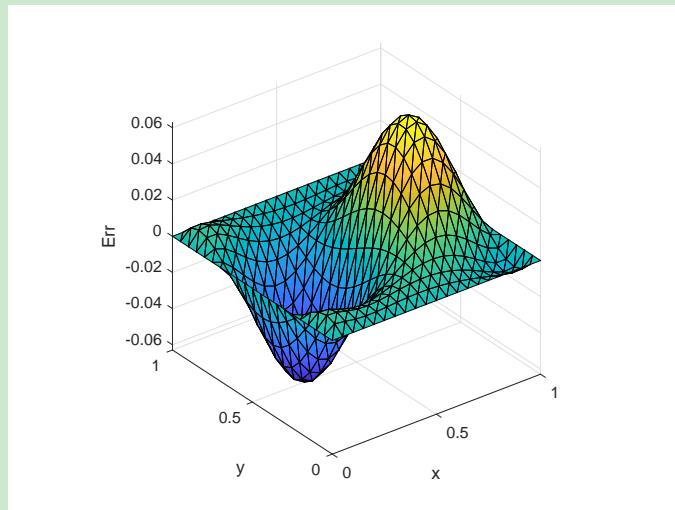


图 7.2. Morley 元方法的绝对误差 ($N_x=N_y=20$)

7.3 非协调 Zienkiewicz 元

Zienkiewicz 元是一种不完全三次三角形元 (Serendipity element).

7.3.1 Zienkiewicz 元的构造

三次插值多项式有 10 个自由度, 一种减少自由度的方法就是直接去掉三次多项式基中的某些项或者某些项共用一个系数.

1. 我们知道三次多项式在面积坐标下可表示成 λ_1, λ_2 和 λ_3 的齐三次式, 即 $\forall P \in \mathbb{P}_3$, 有 $P = \sum_{i+j+k=3} c_{ijk} \lambda_1^i \lambda_2^j \lambda_3^k$. 因形心对应的节点基为 $27\lambda_1\lambda_2\lambda_3$, 故一个例子是在 $P = \sum_{i+j+k=3} a_{ijk} \lambda_1^i \lambda_2^j \lambda_3^k$ 中去掉 $\lambda_1\lambda_2\lambda_3$ 的这一项构造不完全的三次插值.

2. 尽管在边上附加 9 个条件可唯一确定该多项式, 但这样的插值对一次式都不准确. 事实上, 对一次式 $P_1(\lambda) = \lambda_1$, 我们把它表为齐三次式如下

$$\begin{aligned} P_1(\lambda) &= \lambda_1 = \lambda_1(\lambda_1 + \lambda_2 + \lambda_3)^2 \\ &= \lambda_1^3 + \lambda_1\lambda_2^2 + \lambda_1\lambda_3^2 + 2\lambda_1^2\lambda_2 + 2\lambda_1^2\lambda_3 + 2\lambda_1\lambda_2\lambda_3. \end{aligned}$$

这表明为了让一次式精确, 不仅 $\lambda_1\lambda_2\lambda_3$ 不能丢掉, 而且所有含 λ_1 的齐三次单项式都不能去掉. 同理为了让 $P_i(\lambda) = \lambda_i$ 精确, 所有含有 λ_i 的齐三次单项式都不能去掉.

3. 我们的目标是插值的逼近程度尽量高. 三次多项式已不可能, 只好退而求其次, 考虑逼近程度达到二次多项式. 也就是说, 在已有的 9 个自由度的前提下, 再加一个限制条件: 插值的逼近程度达到二次多项式.

给定插值多项式 Q , Zienkiewicz 元构造的条件是

- 给定三角形顶点处的函数值以及两个偏导数值, 即

$$Q(\delta_i) = w_i, \quad Q_x(\delta_i) = w_{xi}, \quad Q_y(\delta_i) = w_{yi}, \quad i = 1, 2, 3.$$

- 插值对二次多项式精确, 即插值多项式包含二次多项式.

经过一些细致的推导, 我们有

$$Q(x, y) = \sum_{i=1}^3 w_i \varphi_i + w_{xi} \psi_i + w_{yi} \zeta_i,$$

其中,

$$\begin{cases} \varphi_1 = \lambda_1^2(3 - 2\lambda_1) + 2\lambda_1\lambda_2\lambda_3, \\ \varphi_2 = \lambda_2^2(3 - 2\lambda_2) + 2\lambda_1\lambda_2\lambda_3, \\ \varphi_3 = \lambda_3^2(3 - 2\lambda_3) + 2\lambda_1\lambda_2\lambda_3, \\ \psi_1 = \lambda_1^2(\xi_1\lambda_2 - \xi_2\lambda_1 - \xi_2) + \left(\frac{1}{2}\xi_1 + \xi_2\right)\lambda_1\lambda_2\lambda_3, \\ \psi_2 = \lambda_2^2(\xi_1\lambda_2 - \xi_2\lambda_1 - \xi_1) + \left(\frac{1}{2}\xi_2 + \xi_1\right)\lambda_1\lambda_2\lambda_3, \\ \psi_3 = \lambda_3^2(\xi_1\lambda_2 - \xi_2\lambda_1) + \left(\frac{1}{2}\xi_1 - \xi_2\right)\lambda_1\lambda_2\lambda_3, \end{cases}$$

$$\begin{cases} \zeta_1 = \lambda_1^2(\eta_1\lambda_2 - \eta_2\lambda_1 - \eta_2) + (\frac{1}{2}\eta_1 + \eta_2)\lambda_1\lambda_2\lambda_3, \\ \zeta_2 = \lambda_2^2(\eta_1\lambda_2 - \eta_2\lambda_1 - \eta_1) + (\frac{1}{2}\eta_2 + \eta_1)\lambda_1\lambda_2\lambda_3, \\ \zeta_3 = \lambda_3^2(\eta_1\lambda_2 - \eta_2\lambda_1) + (\frac{1}{2}\eta_1 - \eta_2)\lambda_1\lambda_2\lambda_3. \end{cases}$$

7.3.2 sparse 装配指标

整体自由度如下排列

$$W = [w_1, \dots, w_N, w_{x1}, \dots, w_{xN}, w_{y1}, \dots, w_{yN}],$$

即先排列所有顶点函数值, 再排顶点处对 x 求偏导的值, 最后排对 y 求偏导的值.

为了单元分析的方便, 单元变量如下排

$$W_\beta = [w_i, w_{xi}, w_{yi}, \quad i = 1, 2, 3]^T,$$

即先排列第一个顶点的自由度, 再排第二个顶点的, 等等. 显然装配指标为

```

1 % ----- Sparse assembling indices -----
2 N = size(node,1); NT = size(elem,1); Ndof = 9;
3 elem2 = zeros(NT,Ndof);
4 elem2(:,[1,4,7]) = elem;
5 elem2(:,[2,5,8]) = elem + N;
6 elem2(:,[3,6,9]) = elem + 2*N;
7 nnz = NT*Ndof^2;
8 ii = zeros(nnz,1); jj = zeros(nnz,1);
9 id = 0;
10 for i = 1:Ndof
11     for j = 1:Ndof
12         ii(id+1:id+NT) = elem2(:,i);
13         jj(id+1:id+NT) = elem2(:,j);
14         id = id + NT;
15     end
16 end

```

7.3.3 双线性形式第一项的计算

这个计算稍微有点复杂, 我们详细说明一下.

考虑如下分块

$$N = [N_1, N_2, N_3], \quad N_i = [\varphi_i, \psi_i, \zeta_i],$$

则插值多项式为

$$Q = NW_\beta,$$

于是

$$K = - \begin{bmatrix} \partial_{11}w \\ \partial_{22}w \\ 2\partial_{12}w \end{bmatrix} \approx - \begin{bmatrix} \partial_{11}N \\ \partial_{22}N \\ 2\partial_{12}N \end{bmatrix} W_\beta =: BW_\beta,$$

式中,

$$B = - \begin{bmatrix} \partial_{11}N \\ \partial_{22}N \\ 2\partial_{12}N \end{bmatrix} = [B_1, B_2, B_3],$$

而

$$B_i = - \begin{bmatrix} \partial_{11}N_i \\ \partial_{22}N_i \\ 2\partial_{12}N_i \end{bmatrix} = - \begin{bmatrix} \partial_{11}\varphi_i & \partial_{11}\psi_i & \partial_{11}\zeta_i \\ \partial_{22}\varphi_i & \partial_{22}\psi_i & \partial_{22}\zeta_i \\ 2\partial_{12}\varphi_i & 2\partial_{12}\psi_i & 2\partial_{12}\zeta_i \end{bmatrix}, \quad i = 1, 2, 3.$$

这样, 单元刚度矩阵为

$$K_\beta = \int_\beta B^T R B dxdy = (K_{st})_{3 \times 3},$$

其中,

$$K_{st} = \int_\beta B_s^T R B_t dxdy.$$

显然, 被积矩阵的每个分量都形为 $f(\lambda_1, \lambda_2, \lambda_3)$, 从而可用三角形上的 Gauss 求积计算.

为了计算 B_i , 我们要计算基函数的二阶导. 从前面基函数的表达式可以看到, 这归结于计算三次齐次式的二阶导

$$\partial_{11}(\lambda_i \lambda_j \lambda_k), \quad \partial_{22}(\lambda_i \lambda_j \lambda_k), \quad \partial_{12}(\lambda_i \lambda_j \lambda_k), \quad 1 \leq i, j, k \leq 3$$

以及 λ_i^2 的二阶导.

1. 我们考虑一般情形的二次齐次式 $\lambda_i \lambda_j$. 直接计算可知, $\partial_{\alpha\beta}(\lambda_i \lambda_j)$ 由两部分组成

- 对同一个求二阶导

$$(\partial_{\alpha\beta} \lambda_i) \lambda_j + \lambda_i (\partial_{\alpha\beta} \lambda_j);$$

- 对不同位置求导

$$(\partial_\beta \lambda_i) (\partial_\alpha \lambda_j) + (\partial_\alpha \lambda_i) (\partial_\beta \lambda_j).$$

注意到 λ_i^2 是一次多项式, 第一部分为零, 故

$$\partial_{\alpha\beta}(\lambda_i\lambda_j) = (\partial_\beta\lambda_i)(\partial_\alpha\lambda_j) + (\partial_\alpha\lambda_i)(\partial_\beta\lambda_j).$$

归结为 λ_i 的一阶导的计算.

2. 再考虑三次齐次式. $\partial_{\alpha\beta}(\lambda_i\lambda_j\lambda_k)$ 的和项由如下部分组成

- 对同一个求二阶导

$$(\partial_{\alpha\beta}\lambda_i)\lambda_j\lambda_k + \lambda_i(\partial_{\alpha\beta}\lambda_j)\lambda_k + \lambda_i\lambda_j(\partial_{\alpha\beta}\lambda_k);$$

- 对 1,2 位置求导

$$(\partial_\alpha\lambda_i)(\partial_\beta\lambda_j)\lambda_k + (\partial_\beta\lambda_i)(\partial_\alpha\lambda_j)\lambda_k;$$

- 对 2,3 位置求导

$$\lambda_i(\partial_\alpha\lambda_j)(\partial_\beta\lambda_k) + \lambda_i(\partial_\beta\lambda_j)(\partial_\alpha\lambda_k);$$

- 对 3,1 位置求导

$$(\partial_\alpha\lambda_i)\lambda_j(\partial_\beta\lambda_k) + (\partial_\beta\lambda_i)\lambda_j(\partial_\alpha\lambda_k).$$

类似地, 第一部分为零. 这样, 它们的二阶导又归结为 λ_i 的一阶导的计算. 易知有

$$\partial_1\lambda_i = \frac{\eta_i}{2S}, \quad \partial_2\lambda_i = -\frac{\xi_i}{2S}, \quad i = 1, 2, 3,$$

它们都是常数.

先计算面积坐标函数的一阶导. 这在前面章节的讨论中已经给出了, 函数为 gradbasis.m, 它给出所有单元的结果且以三维数组存储 (事实上还输出了面积), 形式如下

```

1 Dphi(1:NT,:,:,1) = grad1;
2 Dphi(1:NT,:,:,2) = grad2;
3 Dphi(1:NT,:,:,3) = grad3;

```

它的 3 个部分都是 $NT \times 2$ 的矩阵, 该矩阵的第一列存储 ∂_1 的结果, 第二列则为 ∂_2 的.

接着计算

$$\partial_{\alpha\beta}(\lambda_i\lambda_j) = (\partial_\beta\lambda_i)(\partial_\alpha\lambda_j) + (\partial_\alpha\lambda_i)(\partial_\beta\lambda_j).$$

它显然为常数, 用函数 Dquad.m 计算给定 (i, j) 的所有导数对的结果. 考虑到积分是按单元循环计算的, 该函数只返回固定单元的.

```

1 function D = Dquad(i,j,iel,Dphi,lambda)
2
3 Deriv = @(s,t) Dphi(iel,s,i).*Dphi(iel,t,j) ...
4     + Dphi(iel,s,j).*Dphi(iel,t,i);
5
6 D = [Deriv(1,1),Deriv(2,2),Deriv(1,2)];
7 nI = size(lambda,1);
8 D = repmat(D,nI,1);

```

这里为了方便, 对应积分点给出.

接着计算所有三次齐次式的二阶导在积分点处的值. 为了方便, 我们定义一个函数 Dtriple.m. 它的输入是齐次式的下标 (i, j, k) , 返回的是三个二阶导在积分点处的值. 用矩阵 Dtri 存储, 其维数为 `zeros(nI, 3)` 的矩阵, 其中 nI 是积分点的个数.

```

1 function Dtri = Dtriple(i,j,k,iel,Dphi,lambda)
2
3 Deriv = @(s,t) (Dphi(iel,s,i).*Dphi(iel,t,j) ...
4     +Dphi(iel,t,i).*Dphi(iel,s,j))*lambda(:,k) ...
5     + (Dphi(iel,s,j).*Dphi(iel,t,k) ...
6     +Dphi(iel,t,j).*Dphi(iel,s,k))*lambda(:,i) ...
7     + (Dphi(iel,s,i).*Dphi(iel,t,k) ...
8     +Dphi(iel,t,i).*Dphi(iel,s,k))*lambda(:,j);
9
10 % nI = size(lambda,1); % number of integration points
11 % Dtri = zeros(nI,3); % (11,22,12)
12 Dtri = [Deriv(1,1), Deriv(2,2), Deriv(1,2)];

```

有了 `Dtri` 就可计算固定单元基函数的二阶导数, 这里直接计算 B_i . 考虑到积分点有 nI 个, 我们用三维数组存储, 每个块对应一个积分点.

第八章 多重网格法的算法介绍

本文考虑迭代法中的多重网格法 (the multigrid methods, MG methods), 且只讨论线性元.

8.1 嵌套有限元

8.1.1 嵌套有限元空间与子空间方程

设 Ω 是多角形区域. \mathcal{T}_1 是区域 Ω 的一个三角剖分, 网格参数为 h_1 . 将剖分进行二分, 如将三角形各边的中点进行连接得加细剖分 \mathcal{T}_2 , 相应的网格参数为 $h_2 = h_1/2$. 依此类推, 可获得一组嵌套的剖分

$$\mathcal{T}_1, \mathcal{T}_2 \cdots, \mathcal{T}_L.$$

对每个三角剖分 \mathcal{T}_l , 定义相应的连续分片线性有限元空间为

$$V_l = \{v_l \in C(\bar{\Omega}) : v_l|_K \in \mathbb{P}_1(K), v_l|_{\partial\Omega} = 0\}.$$

显然

$$V_1 \subset V_2 \subset \cdots \subset V_L =: V_h,$$

注意, 这里的 V_l 实际上是有限元空间 V_h 的粗化.

在 l 层上, 有限元方法为: 找 $u_l \in V_l$ 使得

$$a(u_l, v_l) = (f, v_l), \quad v_l \in V_l. \quad (8.1)$$

在空间 V_l 定义算子

$$A_l : V_l \rightarrow V_l, \quad v_l \mapsto A_l v_l,$$

满足

$$(A_l v_l, w_l) = a(v_l, w_l), \quad v_l, w_l \in V_l.$$

定义 f 在 V_l 上的 L^2 投影 $f_l \in V_l$ 为

$$(f_l, v_l) = (f, v_l), \quad v_l \in V_l,$$

则 (8.1) 可写为

$$A_l u_l = f_l.$$

设 V_l 的节点基为 $N_l = (\varphi_1, \dots, \varphi_{n_l})$, 算子 A_l 在该组基下的矩阵为 \mathbf{A}_l , 即

$$A_l N_l = A_l(\varphi_1, \dots, \varphi_{n_l}) = (\varphi_1, \dots, \varphi_{n_l}) \mathbf{A}_l = N_l \mathbf{A}_l.$$

考虑展开

$$u_l = N_l \mathbf{u}_l, \quad f_l = N_l \mathbf{f}_l,$$

则有

$$\mathbf{A}_l \mathbf{u}_l = \mathbf{f}_l.$$

显然, 上式就是有限元方程组. 在不至误会时, 我们仍用不加粗的符号表示.

注 8.1 在最细的网格空间 V_h 上, 记 $A = A_L$, 即

$$(Av_h, w_h) = a(v_h, w_h), \quad v_h, w_h \in V_h.$$

显然有

$$(A_l v_l, w_l) = a(v_l, w_l) = (Av_l, w_l), \quad v_l, w_l \in V_l. \quad (8.2)$$

8.1.2 延长、限制算子与延长、限制矩阵

设

$$V_1 \subset V_2 \subset \cdots \subset V_L$$

为嵌套空间. 定义自然嵌入

$$I_i : V_i \rightarrow V_{i+1}, \quad v_i \mapsto I_i v_i = v_i,$$

有时候也记为 $I_i = I_i^{i+1}$, 称为延长算子. 定义 L^2 投影

$$Q_i : V_{i+1} \rightarrow V_i, \quad v_{i+1} \mapsto Q_i v_{i+1},$$

满足

$$(Q_i v_{i+1}, w_i) = (v_{i+1}, w_i), \quad v_{i+1} \in V_{i+1}, \quad w_i \in V_i,$$

称其为限制算子, 也记为 $Q_i = I_{i+1}^i$. 后者记号即为转移算子.

可以证明, $Q_i = I_i^\top$, 其中 I_i^\top 表示在内积 (\cdot, \cdot) 下的伴随. 事实上,

$$(Q_i^\top v_i, v) = (v_i, Q_i v) = (v_i, v) = (I_i v_i, v) \quad \forall v_i \in V_i, v \in V_{i+1}.$$

定义 8.1 设 $N_i = (\varphi_1, \dots, \varphi_{n_i})$ 是 V_i 的节点基向量, 由 $I_i \varphi_j \in V_{i+1}$ 知, 存在矩阵 \mathbf{I}_i^{i+1} 使得

$$I_i N_i = N_{i+1} \mathbf{I}_i^{i+1}. \quad (8.3)$$

同理, 由 $Q_i \varphi_j \in V_i$, 存在矩阵 \mathbf{I}_{i+1}^i 使得

$$Q_i N_{i+1} = N_i \mathbf{I}_{i+1}^i. \quad (8.4)$$

称 \mathbf{I}_i^{i+1} 为延长矩阵, \mathbf{I}_{i+1}^i 为限制矩阵.

延长矩阵和限制矩阵用以沟通转移前后的节点值.

先考虑延长矩阵. 给定 $v \in V_i$, 它在 V_i 中的节点值向量记为 \mathbf{v}_i , $I_i v = v$ 在 V_{i+1} 中的节点值向量记为 $\hat{\mathbf{v}}_{i+1}$. 注意 $\mathcal{T}_i \subset \mathcal{T}_{i+1}$, 有 \mathbf{v}_i 是 $\hat{\mathbf{v}}_{i+1}$ 的分量. 而 $\hat{\mathbf{v}}_{i+1}$ 其余的分量是三角形边中点对应的函数值, 由线性插值知, 它们可由三角形顶点值平均得到. 于是, 存在矩阵 \mathbf{I}_i^{i+1} 使得

$$\hat{\mathbf{v}}_{i+1} = \mathbf{I}_i^{i+1} \mathbf{v}_i,$$

这里的矩阵就是前面定义的延长矩阵. 事实上, (8.3) 两边作用于 \mathbf{v}_i 有

$$I_i N_i \mathbf{v}_i = N_{i+1} \mathbf{I}_i^{i+1} \mathbf{v}_i \Rightarrow I_i v = N_{i+1} \mathbf{I}_i^{i+1} \mathbf{v}_i,$$

而 $I_i v = N_{i+1} \hat{\mathbf{v}}_{i+1}$, 即证.

再考虑限制矩阵. 给定 $r \in V_{i+1}$, 它在 V_{i+1} 中的节点值向量记为 \mathbf{r}_{i+1} . 因 $Q_i r \in V_i$, 设它在 V_i 中的节点值向量记为 $\tilde{\mathbf{r}}_i$, 则有

$$\tilde{\mathbf{r}}_i = \mathbf{I}_{i+1}^i \mathbf{r}_{i+1},$$

这只需要把 (8.4) 两边作用于 \mathbf{r}_{i+1} .

可以证明,

$$\mathbf{I}_{i+1}^i = (\mathbf{I}_i^{i+1})^\top.$$

8.1.3 Galerkin 条件

对两层情形, 即 $V_1 \subset V_2$, 视 V_2 为 V , 由 (8.2),

$$(A_1 u_1, v_1) = (A u_1, v_1) = (A I_1 u_1, I_1 v_1) = (I_1^\top A I_1 u_1, v_1), \quad (8.5)$$

这表明

$$A_1 = I_1^\top A I_1 = Q_1 A I_1,$$

即

$$A_1 = I_1^\top A_2 I_1 = Q_1 A_2 I_1,$$

其中, $Q_1 = I_2^1$, $I_1 = I_1^2$. 相应的矩阵形式为

$$\mathbf{A}_1 = \mathbf{I}_2^1 \mathbf{A}_2 \mathbf{I}_1^2.$$

类似有

$$\mathbf{A}_i = \mathbf{I}_{i+1}^i \mathbf{A}_{i+1} \mathbf{I}_i^{i+1}. \quad (8.6)$$

推导 (8.5) 用到双线性形式, 因而称 (8.6) 为 Galerkin 条件, 而获得 \mathbf{A}_i 的方法称为 Galerkin 方法或变分方法.

8.2 MG 的基本思想

迭代法通常有两种构造思路, 一是矩阵分裂, 一是残差或误差校正. MG 基于后者.

8.2.1 残差校正与频率分量

考虑线性算子方程

$$Au = f \quad (8.7)$$

的求解. 设 v 是 u 的一个近似, 则误差为 $e = u - v$, 残差为 $r = f - Av$. 显然误差与残差满足

$$Ae = r, \quad (8.8)$$

这样对近似解通过求解方程 (8.8) 可获得校正 $u = v + e$. 为此给出从 u_k 获得 u_{k+1} 的残差校正格式

残差校正算法

- (1) 形成残差: $r = f - Au_k$;
 - (2) 近似求解残差方程 $Ae = r$ 得校正: $\hat{e} = Br$, 其中, B 是 A^{-1} 的近似;
 - (3) 更新: $u_{k+1} = u_k + \hat{e}$.
-

算子 B 称为迭代子 (iterator) 或光滑子 (smoother). 基于残差校正的方法就是给出 A^{-1} 的一个合理近似 B , 从而可以迭代求解

$$u_{k+1} = u_k + \hat{e} = u_k + Br = u_k + B(f - Au_k).$$

对多重网格法, 我们记 $MG := B$, 从而

$$u_{k+1} = u_k + MG(f - Au_k).$$

考虑问题

$$\begin{cases} -u''(x) = f(x), & x \in (0, 1), \\ u(0) = u_0, \quad u(1) = u_1. \end{cases}$$

使用步长为 $h = \frac{1}{N+1}$ 的均匀剖分, 易知线性元的刚度矩阵 (`kk(freeNode, freeNode)`) 为

$$A = \frac{1}{h} \begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & \ddots & & \\ & \ddots & \ddots & -1 & \\ & & & -1 & 2 \end{bmatrix}_{N \times N},$$

其特征值为

$$\lambda_k = \frac{4}{h} \sin^2 \frac{\theta_k}{2}, \quad k = 1, 2, \dots, N,$$

相应特征向量为

$$\mathbf{p}_k = [\sin(\theta_k), \sin(2\theta_k), \dots, \sin(N\theta_k)]^T, \quad 1 \leq k \leq N,$$

式中 $\theta_k = \frac{k}{N+1}\pi$, $1 \leq k \leq N$. 特征向量满足如下的正交关系

$$\mathbf{p}_i^T \mathbf{p}_j = \frac{1}{2h} \delta_{ij},$$

且任何一个 N 维向量都可由它展开, 自然误差和残差也是如此.

定义 8.2 称特征向量

$$\mathbf{p}_k = [\sin(\theta_k), \sin(2\theta_k), \dots, \sin(N\theta_k)]^T$$

为低频分量, 如果 $1 \leq k < \lfloor \frac{N}{2} \rfloor$. 剩下的则称为高频分量.

对小的 k , 正弦函数的波数少, 函数图像更加光滑, 所以低频分量也称为光滑分量. 高频分量函数图像摆动较大, 也称为摆动分量.

8.2.2 经典迭代法对频率分量的影响

为了方便, 考虑方程组 $Au = f$, 其中

$$A = \begin{bmatrix} 2 & -1 & & \\ -1 & 2 & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 2 \end{bmatrix}.$$

设 A 分裂为 $A = D - L - U$, 其中 D 是对角线元素给出的对角矩阵, $-L$ 是下三角部分, $-U$ 是上三角部分. Jacobi 迭代可写为

$$Du^{(n+1)} = (L + U)u^{(n)} + f$$

或

$$u^{(n+1)} = D^{-1}(L + U)u^{(n)} + D^{-1}f =: Bu^{(n)} + D^{-1}f,$$

式中,

$$B = \frac{1}{2} \begin{bmatrix} 0 & 1 & & \\ 1 & 0 & \ddots & \\ & \ddots & \ddots & 1 \\ & & 1 & 0 \end{bmatrix}.$$

显然精确解 u 满足上面的迭代方程, 定义误差 $e^{(n)} = u^{(n)} - u$, 则有

$$e^{(n+1)} = Be^{(n)} = B^{n+1}e^{(0)}.$$

矩阵 B 的特征向量与 A 相同, 而特征值为

$$\lambda_k = 1 - 2\sin^2 \frac{\theta_k}{2}, \quad \theta_k = \frac{k\pi}{N+1}.$$

设初始误差展开为

$$e^{(0)} = a_1 \mathbf{p}_1 + a_2 \mathbf{p}_2 + \cdots + a_N \mathbf{p}_N,$$

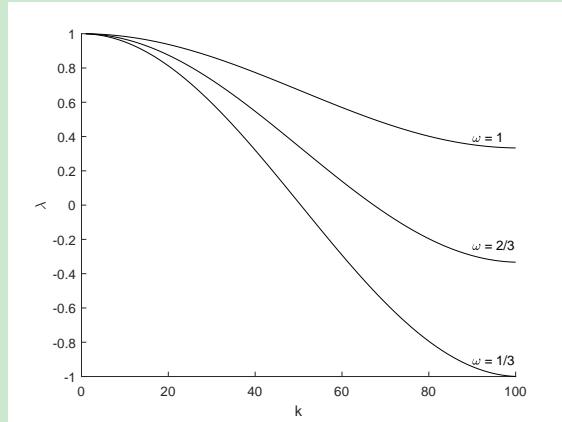
则有

$$e^{(n)} = B^n e^{(0)} = \sum_{i=1}^n a_i \lambda_i^n \mathbf{p}_i.$$

可以看到, 特征值是误差分量的放大系数. 对这里的问题, 显然有 $|\lambda_k| \leq 1$, 从而迭代收敛.

若考虑阻尼 Jacobi 迭代, 设松弛因子为 ω , 则有

$$\lambda_k(\omega) = 1 - 2\omega \sin^2 \frac{\theta_k}{2}, \quad \theta_k = \frac{k\pi}{N+1}.$$



上图给出的是不同阻尼系数特征值随波数 k 的变化. 可以看到, 不论如何选取阻尼系数 ω , 低频分量的衰减系数都不会很小. 这就是为什么经典迭代法在开始衰减较快而后面几乎停滞不动的原因. 事实上, 迭代法最开始高频分量迅速衰减, 而剩下的低频分量衰减并不明显.

注 8.2 残差校正是给定 u_k 获得 u_{k+1} . 这个过程中的迭代初值 $u^{(0)} = u_k$, 残差

$$r = f - Au_k = f - Au^{(0)} = A(u - u^{(0)}) = -Ae^{(0)}.$$

消除误差 $e^{(0)} = u - u_k$ 的频率分量可等价地说消去残差 r 的频率分量.

8.2.3 两网格方法

多重网格法基于如下观察

- 误差分为高频分量和低频分量;
- 经典迭代法可消除误差的高频分量, 对低频分量的抑制作用并不明显;
- 低频部分在粗网格上就可获得较好的近似;
- 细网格上导出的方程条件数更大.

低频或高频与网格剖分有关, 细网格上的一些低频分量在粗网格上是高频分量. MG 的想法就是把细网格上的低频误差视为粗网格上的高频误差, 而对粗网格问题, 我们可再次使用光滑作用. 多重网格法的关键步骤是

- 在细网格上利用通常的迭代法, 如 Richardson 迭代, Gauss-Seidel 迭代进行迭代, 消去残差的高频部分;
- 把细网格上残差的低频部分转移到粗网格上进行残差校正 (视其为粗网格上的高频误差).

根据残差校正方法, 我们是对残差方程进行计算. 一个简单的二层网络如下

算法 3 求解残差方程的两网格算法

1. 前光滑: 在细网格 \mathcal{T}_h 上用某种迭代法求解

$$A_h e_h = r_h,$$

迭代 m_1 次获得近似解 e_h , 从而有残差方程的残差 (仍记为 r_h)

$$r_h \leftarrow r_h - A_h v_h.$$

这里 A_h 对应网格参数 h 的有限元方程.

2. 粗网格校正:

- 将细网格上的残差 r_h 转移到粗网格 \mathcal{T}_{2h} 上, 记为 $I_h^{2h} r_h$.

- 在粗网格上求解残差方程

$$A_{2h} e_{2h} = I_h^{2h} r_h,$$

获得误差 e_{2h} .

- 将粗网格上的误差 e_{2h} 转移到细网格上, 记为 $I_{2h}^h e_{2h}$, 从而获得校正

$$e_h \leftarrow e_h + I_{2h}^h e_{2h}.$$

3. 后光滑: 以 e_h 为初值, 在细网格上用某种迭代法迭代 m_2 次, 获得最终的近似解 e_h .

注 8.3 以下称转移到粗网格上为限制, 而转移到细网格上为延长, 且称开始的残差方程为原方程. 两网格方法的特点是

细网格上: 求解原方程, 即前光滑迭代, 获得残差

↓

粗网格上: 求解残差限制方程, 获得误差

↓

细网格上: 误差延长, 校正原方程的近似解, 并后光滑迭代

注 8.4 两网格方法是直接求解粗网格上的残差限制方程, 它与原方程形式相同, 只不过在粗网格上. 现在可把粗网格视为细网格, 粗网格的下一级粗网格视为粗网格, 而残差限制方程视为原方程, 从而再次使用两网格方法求解. 重复这个思想就获得多重网格法.

8.3 MG 的算法描述

8.3.1 MG 的两网格添加

根据注 8.4, 为了获得多重网格算法, 只需要逐次添加两网格.

- 1 个两网格的过程如下

$$\begin{cases} A_L e_L = r_L \Rightarrow e_L, \quad r_L \leftarrow r_L - A_L e_L \\ A_{L-1} e_{L-1} = Q_{L-1} r_L \Rightarrow e_{L-1}, \\ e_L \Leftarrow e_L + I_L e_{L-1} \end{cases},$$

最后的双箭头包含两层意思: 一是校正, 二是后光滑迭代 (仍用原符号).

- 2 个两网格的过程如下

$$\begin{cases} A_L e_L = r_L \Rightarrow e_L, \quad r_L \leftarrow r_L - A_L e_L \\ A_{L-1} e_{L-1} = Q_{L-1} r_L \left\{ \begin{array}{l} A_{L-1} e_{L-1} = r_{L-1} := Q_{L-1} r_L \Rightarrow e_{L-1}, \quad r_{L-1} \\ A_{L-2} e_{L-2} = Q_{L-2} r_{L-1} \Rightarrow e_{L-2} \\ e_{L-1} \Leftarrow e_{L-1} + I_{L-1} e_{L-2} \end{array} \right. \\ e_L \Leftarrow e_L + I_L e_{L-1} \end{cases},$$

这里视 $Q_{L-1} r_L$ 为 r_{L-1} .

由此可以看到, 在添加两网格的过程中,

- 先逐层递减求残差限制方程 $\begin{cases} Q_{L-1} \\ Q_{L-2}; \\ \vdots \end{cases}$
- 再逐层递增延长误差并后光滑 $\begin{cases} \vdots \\ I_{L-1}. \\ I_L \end{cases}$

具体来说, 多重网格法的计算过程分为如下两步:

算法 4 求解残差方程的多重网格法

Step 1: 计算每层的校正误差: 残差限制与前光滑

- L 层: 给定初值 $e_L^{(0)}$, 迭代求解 $A_L e_L = r_L$, 结果记为 e_L , (新的) 残差仍记为 r_L .
- $L-1$ 层: 以 $e_{L-1}^{(0)} = 0$ 为初值, 迭代求解 $A_{L-1} e_{L-1} = r_{L-1} := Q_{L-1} r_L$, 结果记为 e_{L-1} , 残差为 r_{L-1} .
- $L-2$ 层: 以 $e_{L-2}^{(0)} = 0$ 为初值, 迭代求解 $A_{L-2} e_{L-2} = r_{L-2} := Q_{L-2} r_{L-1}$, 所得结果记为 e_{L-2} , 残差为 r_{L-2} .
- ⋮

Step 2: 延长校正误差与后光滑

⋮

- 对 e_{L-2} 校正: $e_{L-2} \leftarrow e_{L-2} + I_{L-2} e_{L-3}$, 并以校正值为初值, 对 $A_{L-2} e_{L-2} = r_{L-2}$ 做后光滑, 结果仍记为 e_{L-2} .
 - 对 e_{L-1} 校正: $e_{L-1} \leftarrow e_{L-1} + I_{L-1} e_{L-2}$, 并以校正值为初值, 对 $A_{L-1} e_{L-1} = r_{L-1}$ 做后光滑, 结果仍记为 e_{L-1} .
 - 对 e_L 校正: $e_L \leftarrow e_L + I_L e_{L-1}$, 并以校正值为初值, 对 $A_L e_L = r_L$ 做后光滑, 结果记为 e_L , 它就是残差方程的多重网格解.
-

上面的过程可用下图表示

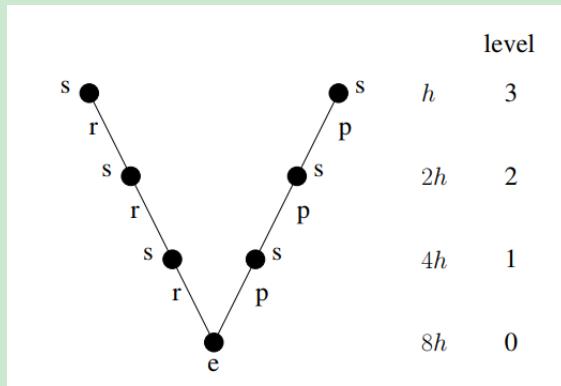


图 8.1. V 循环: s - smoothing, r - restriction, p - prolongation, e - exact solver.

这里规定: 最粗的层直接求解. 因图像形似 V, 称上面的多重网格法为 V 循环多重网格法.

8.3.2 V-循环的伪代码

根据上面的算法, V-循环的伪代码可写为

```

1 function e = Vcycle(r,J)
2 % Solve the residual equation Ae = r by multigrid V-cycle method
3
4 ri = cell(J,1);      % residual in each level
5 ei = cell(J,1);      % correction in each level
6 ri{J} = r;            % initial residual
7
8 % ----- Correction in each level -----
9 for j = J:-1:2
10    % pre-smoothing: one step
11    ei{j} = R{j}*ri{j};
12    % update and restrict residual
13    ri{j-1} = Res{j-1}*(ri{j} - Ai{j}*ei{j});
14 end
15 ei{1} = Ai{1}\ri{1}; % exact solver in the coarsest level
16
17 % ----- prolongation and correction -----
18 for j = 2:J
19    % prolongation and correction
20    ei{j} = ei{j}+Pro{j-1}*ei{j-1};
21    % post-smoothing: one step
22    ei{j} = ei{j} + R{j}'*(ri{j}-Ai{j}*ei{j});
23 end
24 e = ei{J};

```

注 8.5 后光滑中采用 R'_j 即前光滑算子 R_j 的转置, 以保证迭代矩阵 B 是对称的 (这里涉及到迭代格式的对称化, 略). 这样, B 可在预处理共轭梯度 (Preconditioned Conjugate Gradient, PCG) 法中充当预处理子. 正因为如此, 多重网格法本质上是一种预处理方法. 若在前光滑中采用 Gauss-Seidel 迭代, 即 $\mathbf{R}_j = (\mathbf{D}_j + \mathbf{L}_j)^{-1}$, 则后处理中算子转置对应的矩阵为 $(\mathbf{D}_j + \mathbf{U}_j)^{-1}$, 并不是矩阵直接转置. 前者称为 forward Gauss-Seidel, 而后者称为 backward Gauss-Seidel (实际上就是追赶法的两种顺序). 注意, 这里 $\mathbf{A} = \mathbf{D} + \mathbf{L} + \mathbf{U}$.

注 8.6 对 2 个两网格, 在粗网格上, 我们由 e_{L-1}, r_{L-1} 获得新的 e_{L-1} . 这个过程还可以再一次执行. 事实上, 由新的 e_{L-1} 可获得 $r_{L-1} \leftarrow r_{L-1} - A_{L-1}e_{L-1}$, 从而导致需要的 e_{L-1}, r_{L-1} . 这个过程如下

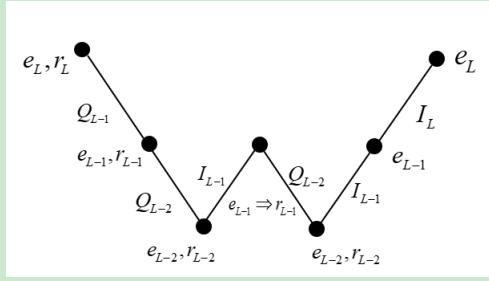


图 8.2. W 循环

该循环形似 W, 称为 W 循环多重网格法. 本文只考虑 V 循环.

8.4 MG 矩阵的获得

暂时考虑一维问题.

8.4.1 延长矩阵

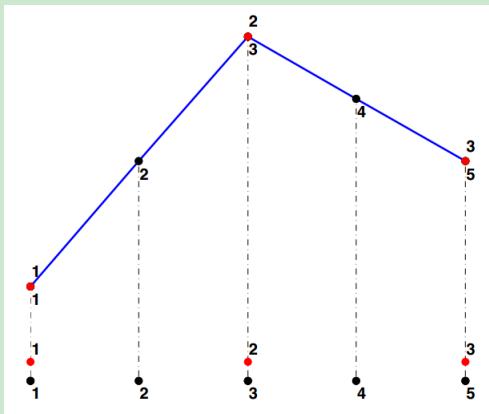


图 8.3. 延长矩阵示意图

如图, 设 3 个红色点对应的粗网格空间为 V_1 , 它是由这些节点值构造的线性有限元空间. 类似地, V_2 是由 5 个黑色点给出的有限元空间. 对 $u_1 \in V_1$, 设节点值向量为 \mathbf{u}_1 . 根据定义, $I_1^2 u_1$ 是 u_1 自然嵌入到 V_2 所得, 它们表示同一个函数, 都是 u_1 . 由 V_1 是线性有限元空间知, 在 V_2 对应的加细节点处, 函数值由 V_1 中的插值获得, 即仍在蓝色直线上. 于是

- 向量 $\mathbf{u}_1 \in \mathbb{R}^3$, $\mathbf{u}_2 = I_1^2 \mathbf{u}_1 \in \mathbb{R}^5$ 对应 V_2 中的同一个函数, 称 \mathbf{u}_2 为延长向量.

对细网格和粗网格的公共节点, 有

$$\mathbf{u}_2(1) = \mathbf{u}_1(1), \quad \mathbf{u}_2(3) = \mathbf{u}_1(2), \quad \mathbf{u}_2(5) = \mathbf{u}_1(3).$$

对细网格上的其他点, 显然延长向量满足

$$\mathbf{u}_2(2) = \frac{\mathbf{u}_1(1) + \mathbf{u}_1(2)}{2}, \quad \mathbf{u}_2(4) = \frac{\mathbf{u}_1(2) + \mathbf{u}_1(3)}{2}.$$

基于以上的式子, 我们有

$$\begin{bmatrix} \mathbf{u}_2(1) \\ \mathbf{u}_2(2) \\ \mathbf{u}_2(3) \\ \mathbf{u}_2(4) \\ \mathbf{u}_2(5) \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 1/2 & 1/2 & 0 \\ 0 & 1 & 0 \\ 0 & 1/2 & 1/2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{u}_1(1) \\ \mathbf{u}_1(2) \\ \mathbf{u}_1(3) \end{bmatrix} \Leftrightarrow \mathbf{u}_2[1, 2, 3, 4, 5] = \mathbf{I}_1^2 \mathbf{u}_1[1, 2, 3], \quad (8.9)$$

右侧方括号中是指标的顺序.

我们把细网格上的节点分为两类:

- \mathcal{C} : 粗细网格的公共节点集, 实际上就是粗网格上的节点集;
- \mathcal{F} : 只在细网格上的节点集, 称为细网格上的额外节点集.

为了方便, 以下在程序中规定: 公共部分, 即粗网格部分用 Coarse 标记, 细网格用 Fine 标记, 而额外的细网格部分用 f 标记.

先考虑 \mathcal{F} . 对图 8.3, \mathcal{F} 对应的编号为 {2, 4}, 节点 2 处的值可由粗网格两侧点处的值获得, 对应粗网格编号 1,2 (所在单元的左右顶点编号). 同理, 4 处对应粗网格左右的编号 2,3. 为此, 我们定义分层基 (hierarchical basis) 矩阵 HB 如下

$$\text{HB} = \begin{bmatrix} 2 & 1 & 2 \\ 4 & 2 & 3 \end{bmatrix},$$

其元素如下

- 第 1 列: 细网格上的额外节点编号 (粗区间中点编号);
- 第 2 列: 额外节点所在粗单元的左顶点编号;
- 第 3 列: 额外节点所在粗单元的右顶点编号.

显然 2,3 列对应第 1 列的插值节点.

对集合 \mathcal{C} , 尽管它们是公共节点, 但在粗细网格上的编号并不相同. 为此, 我们需要定义一个指标映射 `Coarse2Fine`, 它把粗网格上的编号映射为细网格上的. 图 8.3 中给出的

```
Coarse2Fine = [1 3 5]';
```

可利用 `sparse` 命令生成延长矩阵, 如下

```

1 ii = [Coarse2Fine; HB(:,1); HB(:,1)];
2 jj = [CoarseId; HB(:,2); HB(:,3)];
3 ss = [ones(nCoarse,1); 0.5*ones(nf,1); 0.5*ones(nf,1)];
4 Pro = sparse(ii,jj,ss,nFine,nCoarse);

```

简单解释一下:

- 延长矩阵的行 `ii` 对应细网格, 列 `jj` 对应粗网格.
- `ii`, `jj`, `ss` 元素如下

	<code>ii</code>	<code>jj</code>	<code>ss</code>
公共部分	细网格编号	粗网格编号	值为 1
额外部分	额外细节点编号	插值左端点编号	值为 0.5
额外部分	额外细节点编号	插值右端点编号	值为 0.5

- `nCoarse` 是粗网格节点的个数, `nFine` 是细网格节点的个数, `nf` 是细网格中额外点的个数.

有了延长矩阵 `Pro`, 则限制矩阵 `Res = Pro'`, 从而可得每层的刚度矩阵.

8.4.2 延长矩阵的程序实现

给定初始剖分, 我们可生成 `J` 个嵌套剖分 (含初始), 但计算中只需要存储最后一个剖分的 `node`, `elem`.

1. 新剖分通过对上一个剖分的单元添加中点获得, 并接着顶点的序号进行编号. 如下生成最后一个剖分

```

1 a = 0; b = 1;
2 NO = 3; x = linspace(a,b,NO)'; % NO: number of initial nodes
3 node0 = x; elem0 = [(1:NO-1)', (2:NO)']; % initial mesh
4 for j = 2:J
5     node1 = node0; elem1 = elem0;
6     xc = (node1(elem1(:,1))+node1(elem1(:,2)))/2;
7     N = size(node1,1); nel = size(elem1,1); idc = (1:nel)' + N;
8     node = [node1;xc];
9     elem = [[elem1(:,1),idc]; % left subintervals
10           [idc, elem1(:,2)]]; % right subintervals
11    node0 = node; elem0 = elem;

```

```
12 end
```

新的 elem 总是先存储分割的左区间, 再存储右区间. 显然, 此时区域左右端点的编号就是初始剖分对应的编号, 即左端点编号为 1, 右端点编号为 N0.

2. 对上面的循环, 当前情形下, 分层基矩阵如下获得

```
1 % HB in the current level
2 HB = zeros(nel,3);
3 HB(:,1) = idc; HB(:,2:3) = elem1;
```

3. 由于我们是在原有剖分节点的基础上继续编号, 故公共节点处的编号不变, 从而有

```
1 % Prolongation matrix
2 Coarse2Fine = (1:N)'; CoarseId = (1:N)';
3 nCoarse = N; nf = nel; nFine = nCoarse+nf;
4 ii = [Coarse2Fine; HB(:,1); HB(:,1)];
5 jj = [CoarseId; HB(:,2); HB(:,3)];
6 ss = [ones(nCoarse,1); 0.5*ones(nf,1); 0.5*ones(nf,1)];
7 Pro{j-1} = sparse(ii,jj,ss,nFine,nCoarse);
8 Res{j-1} = Pro{j-1}';
```

上面生成了最终的网格剖分、延长矩阵和限制矩阵, 为了方便, 编写为函数 MeshVcycle.m.

```
1 function [node,elem,Pro,Res] = MeshVcycle(node0,elem0,J)
2 %J = 4; % level length, J≥2
3 Pro = cell(J-1,1); Res = cell(J-1,1);
4 for j = 2:J
5     % node and elem in each level
6     node1 = node0; elem1 = elem0;
7     xc = (node1(elem1(:,1))+node1(elem1(:,2)))/2;
8     N = size(node1,1); nel = size(elem1,1); idc = (1:nel)' + N;
9     node = [node1;xc];
10    elem = [[elem1(:,1),idc];      % left subintervals
11              [idc, elem1(:,2)]];    % right subintervals
12    node0 = node; elem0 = elem;
13
14    % HB in the current level
15    HB = zeros(nel,3);
```

```

16     HB(:,1) = idc; HB(:,2:3) = elem1;
17
18 % Prolongation matrix
19 Coarse2Fine = (1:N)'; CoarseId = (1:N)';
20 nCoarse = N; nf = nel; nFine = nCoarse+nf;
21 ii = [Coarse2Fine; HB(:,1); HB(:,1)];
22 jj = [CoarseId; HB(:,2); HB(:,3)];
23 ss = [ones(nCoarse,1); 0.5*ones(nf,1); 0.5*ones(nf,1)];
24 Pro{j-1} = sparse(ii,jj,ss,nFine,nCoarse);
25 Res{j-1} = Pro{j-1}';
26 end

```

8.5 一维问题的 MG 方法

8.5.1 刚度矩阵和载荷向量

在一维问题的有限元方法中, 我们已经给出了说明, 只需要注意此时的区间端点编号为初始网格剖分的, 从而给出边界条件序号.

```

1 % ----- Initial mesh and boundary conditions -----
2 a = 0; b = 1;
3 NO = 3; x = linspace(a,b,NO)'; % NO: number of initial nodes
4 node0 = x; elem0 = [(1:NO-1)', (2:NO)']; % initial mesh
5 J = 4; % level length, J≥2
6
7 Neumann = 1; Dirichlet = NO;
8 bdStruct = struct('Dirichlet', Dirichlet, 'Neumann', Neumann);

```

其他部分只要稍加修改即可. 直接法求解只考虑自由节点部分, 即

```

1 u(freeNode) = kk(freeNode,freeNode)\ff(freeNode); % direct

```

MG 方法则要整体考虑, 因为新增的点要用到边界值进行平均. 为此, 要保留 Dirichlet 节点变量, 做法就是直接恒等替换 (并保持对称), 如下

```

1 u = zeros(N,1); u(bdNode) = g_D(node(bdNode));
2 ff = ff - kk*u;
3 % -----
4 A = eye(N); A(freeNode,freeNode) = kk(freeNode,freeNode);
5 b = u; b(freeNode) = ff(freeNode);

```

这里的 A, b 就是 MG 求解的矩阵和右端.

完整的程序为

```
1 function u = FEM1DVcycle(node,elem,pde,bdStruct,Pro,Res)
2 %FEM1DVcycle solves the 1-D partial diffential equation by Multigrid
3 % V-cycle method
4 %
5 % au' + bu + cu = f, x \in (x0, xL);
6 % g_N = du(x0), g_D = u(xL);
7 % or g_D = u(x0), g_N = du(xL);
8 %
9 % where, g_N for Neumann boundary conditions and
10 % g_D for Dirichlet boundary conditions,
11 % a, b and c are constants.
12 %
13 % Copyright (C) Terence YUE Yu.
14
15 N = size(node,1); nel = size(elem,1); Ndof = 2;
16 % ----- Sparse assembling indices -----
17 nnz = nel*Ndof^2;
18 ii = zeros(nnz,1); jj = zeros(nnz,1);
19 id = 0;
20 for i = 1:Ndof
21     for j = 1:Ndof
22         ii(id+1:id+nel) = elem(:,i); % zi
23         jj(id+1:id+nel) = elem(:,j); % zj
24         id = id + nel;
25     end
26 end
27
28 % ----- Assemble stiffness matrix -----
29 % All element matrices
30 para = pde.para;
31 acoef = para.acoef; bcoef = para.bcoef; ccoef = para.ccoef;
32 x1 = node(elem(:,1)); x2 = node(elem(:,2));
33 h = x2-x1;
34 k11 = -acoef./h+bcoef/2*(-1)+ccoeff.*h./6*2;
35 k12 = -acoef./h*(-1)+bcoef/2+ccoeff.*h./6;
36 k21 = -acoef./h*(-1)+bcoef/2*(-1)+ccoeff.*h./6;
37 k22 = -acoef./h+bcoef/2+ccoeff.*h./6*2;
```

```

38 K = [k11,k12,k21,k22]; % stored in rows
39 % stiffness matrix
40 kk = sparse(ii,jj,K(:,N,N));
41
42 % ----- Assemble load vector -----
43 xc = (x1+x2)./2;
44 F1 = pde.f(xc).*h./2; F2 = F1; F = [F1,F2];
45 ff = accumarray(elem(:), F(:,[N 1]));
46
47 Neumann = bdStruct.Neumann;
48 Dirichlet = bdStruct.Dirichlet;
49 % ----- Neumann boundary conditions -----
50 g_N = pde.g_N;
51 if isempty(Neumann)
52     NO = max(Neumann,Dirichlet);
53     bn = zeros(N,1); bn(1) = -1; bn(NO) = 1; % -1: left norm vector
54     bn(Neumann) = bn(Neumann)*g_N(node(Neumann));
55     ff(Neumann) = ff(Neumann) + (-acoef)*bn(Neumann);
56 end
57
58 % ----- Dirichlet boundary conditions -----
59 Dirichlet = bdStruct.Dirichlet; g_D = pde.g_D;
60 isBdNode = false(N,1); isBdNode(Dirichlet) = true;
61 bdNode = find(isBdNode); freeNode = find(~isBdNode);
62 u = zeros(N,1); u(bdNode) = g_D(node(bdNode));
63 ff = ff - kk*u;
64
65 % -----
66 %u(freeNode) = kk(freeNode,freeNode)\ff(freeNode); % direct
67 A = eye(N); A(freeNode,freeNode) = kk(freeNode,freeNode);
68 b = u; b(freeNode) = ff(freeNode);
69 u = mgVcycle1D(A,b,Pro,Res); % multigrid Vcycle

```

8.5.2 多重网格函数

mgVcycle1D 函数

前面给出了 V-循环的伪代码, 用到延长矩阵、限制矩阵, 每层的系数矩阵以及光滑迭代. 有了 Pro, Res, 就可生成每层的矩阵 Ai. mgVcycle1D.m 函数如下

```

1 function u = mgVcycle1D(A,b,Pro,Res)
2
3 J = length(Pro)+1;
4 Ai = cell(J,1); Ai{J} = A; % matrices in subspaces
5 for j = J-1:-1:1
6     Ai{j} = Res{j}*Ai{j+1}*Pro{j};
7 end
8
9 tol = 1e-6; tolf = tol * norm(b); % Relative tolerance
10 Err = 10; u = zeros(size(b));
11 iter = 0; MaxIt = 20;
12 while Err>tolf && iter≤MaxIt
13     r = b-A*u;
14     e = Vcycle1D(A,r,Ai,Pro,Res);
15     u = u+e;
16     Err = norm(e); iter = iter + 1;
17 end

```

Vcycle1D 函数

现在伪代码可如下具体化

```

1 function e = Vcycle1D(A,r,Ai,Pro,Res)
2 % Solve the residual equation Ae = r by multigrid V-cycle method
3
4 J = length(Ai); % level length
5
6 % If the problem is small enough, solve it directly
7 if J≤1
8     e = A\r;      return;
9 end
10
11 ri = cell(J,1);          % residual in each level
12 ei = cell(J,1);          % correction in each level
13 ri{J} = r;
14
15 % ----- Correction in each level -----
16 option = 'forward';
17 for j = J:-1:2
18     % % pre-smoothing: one step

```

```

19      % ei{j} = R{j}*ri{j};
20      ei{j} = smoother(Ai{j},ri{j},option);
21      % update and restrict residual
22      ri{j-1} = Res{j-1}*(ri{j}-Ai{j}*ei{j});
23 end
24 ei{1} = Ai{1}\ri{1}; % exact solver in the coarsest level
25
26 % ----- prolongation and correction -----
27 option = 'backward';
28 for j = 2:J
29     % prolongation and correction
30     ei{j} = ei{j}+Pro{j-1}*ei{j-1};
31     % % post-smoothing: one step
32     % ei{j} = ei{j} + R{j}'*(ri{j}-Ai{j}*ei{j});
33     rij = ri{j}-Ai{j}*ei{j};
34     ei{j} = ei{j} + smoother(Ai{j},rij,option);
35 end
36 e = ei{J};

```

注意, 这个程序是抽象的, 它对一般情形也是正确的.

smoother 函数

前后光滑分别用 forward Gauss-Seidel 和 backward Gauss-Seidel, 程序如下

```

1 function ei = smoother(Ai,ri,option)
2 switch option
3     case 'forward'
4         Ri = tril(Ai); % Forward Gauss-Seidel    R = D+L
5     case 'backward'
6         Ri = triu(Ai); % Backward Gauss-Seidel   R = D+U
7 end
8 ei = Ri\ri;

```

8.5.3 数值结果

主程序如下

```

1 clc;clear;close all
2 % ----- Initial mesh and boundary conditions -----
3 a = 0; b = 1;

```

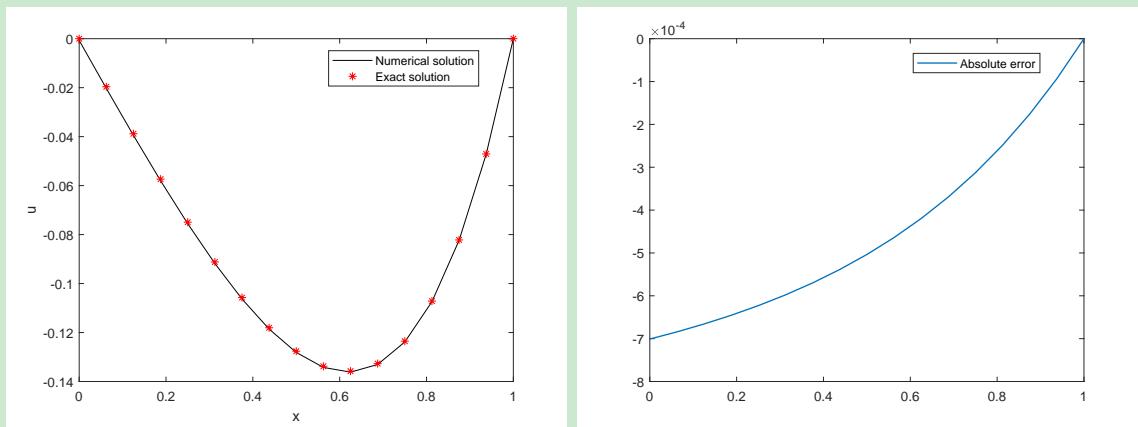
```

4 NO = 3; x = linspace(a,b,NO)'; % NO: number of initial nodes
5 node0 = x; elem0 = [(1:NO-1)', (2:NO)']; % initial mesh
6 J = 4; % level length, J≥2
7
8 Neumann = 1; Dirichlet = NO;
9 bdStruct = struct('Dirichlet', Dirichlet, 'Neumann', Neumann);
10
11 % ----- mesh and prolongation -----
12 [node,elem,Pro,Res] = MeshVcycle(node0,elem0,J);
13
14 % ----- PDE -----
15 acoef = -1; bcoef = 0; ccoef = 0;
16 para = struct('acoef', acoef, 'bcoef', bcoef, 'ccoeff', ccoef);
17 pde = pdedata1D(para);
18
19 % ----- FEM1DVcycle -----
20 u = FEM1DVcycle(node,elem,pde,bdStruct,Pro,Res);
21
22 % ----- error analysis -----
23 [node,id] = sort(node);
24 u = u(id);
25 uexact = pde.uexact(node);
26 figure, plot(node,u,'k',node,uexact,'r*');
27 xlabel('x'); ylabel('u');
28 legend('Numerical solution','Exact solution')
29 Err = u-uexact;
30 figure, plot(node,Err,'linewidth',1); legend('Absolute error');

```

注意, 前面生成的 `node` 不是按坐标顺序给出的, 直接画图会出现节点交错连接. 为了避免这一点, 我们重新排序, 或者类似二维逐个单元画图 (回忆二维 patch).

计算结果如下



(a) 数值解

(b) 绝对误差

图 8.4. 数值解与绝对误差 ($J = 4$)

参考文献